TAMPERE UNIVERSITY OF TECHNOLOGY

SARATH SINGAPATI

INTER PROCESS COMMUNICATION IN ANDROID

MASTER OF SCIENCE THESIS

Examiner: Professor Tommi Mikkonen
Examiner and thesis subject approved by
The Faculty of Computing and Electrical
Engineering on 7th March 2012

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
SARATH SINGAPATI
INTER PROCESS COMMUNICATION IN ANDROID
Master of Science Thesis, 45 pages, 4 Appendix pages
June 2012
Major: Software Systems
Examiner: Professor Tommi Mikkonen
Keywords: Android, Google, mobile applications, Process, IPC

Google's Android mobile phone software platform is currently the big opportunity for application software developers. Android has the potential for removing the barriers to success in the development and sale of a new generation of mobile phone application software. Just as the standardized PC and Macintosh platforms created markets for desktop and server software, Android, by providing a standard mobile phone application environment, creates a market for mobile applications and the opportunity for applications developers to profit from those applications.

One of the main intentions of Android platform is to eliminate the duplication of functionality in different applications to allow functionality to be discovered and invoked on the fly, and to let users replace applications with others that offer similar functionality. The main problem here is how to develop applications that must have as few dependencies as possible, and must be able to provide services to other applications.

This thesis studies the Android mobile operating system, its capabilities in developing applications that communicate with each other and provide services to other applications. As part of the study, a sample application called "Event Planner", has been developed to experiment how Inter Process Communication works in Android platform, explains how to implement, and use Inter Process Communication (IPC).

In summary, this thesis suggests that inter process communication is very useful mechanism in Android programming model to provide flexible, efficient message passing between processes, and platform has all the needed tools, operating system components to implement IPC.

PREFACE

This thesis work was done between October 2011 and May 2012. I would like to thank my examiner Prof. Tommi Mikkonen for guiding me throughout this thesis work.

I would like to thank Mikko Hartikainen for reading the thesis text and providing comments.

I thank my friends, classmates for their help and encouragement during my studies. Finally, I would like to thank my parents and my wife for their support throughout my studies.

Tampere, 21.05.2012

Sarath Singapati
sarath.singapati@gmail.com

CONTENTS

ABBREVIATIONS

AAC	Advanced Audio Coding
AAPT	Android Asset Packaging Tool
ADB	Android Debug Bridge
ADT	Android Development Tools
AIDL	Android Interface Description Language
AMR	Adaptive Multi-Rate
API	Application Programming Interface
APK	Android Package File
AVD	Android Virtual Device
BSD	Berkeley Software Distribution
DDMS	Dalvik Debug Monitor Server
DEX	Dalvik Executable
DVM	Dalvik Virtual Machine
GIF	Graphics Interchange Format
GPS	Global Positioning System
IDE	Integrated Development Environment
IM	Instant Messaging
IPC	Inter Process Communication
JPG	Joint Photographic Experts Group
LCD	Liquid Crystal Display
LiMo	Linux Mobile
LRU	Least Recently Used
MP3	MPEG-1 Audio Layer 3
MPEG	Moving Picture Experts Group
OHA	Open Handset Alliance
OOM	Out Of Memory
OS	Operating System
PNG	Portable Network Graphics
RIM	Research In Motion
RPC	Remote Procedure Call
SDK	Software Development Kit
SGL	Scalable Graphics Library
SMS	Short Message Service
SQL	Structured Query Language
UI	User Interface
UML	Unified Modeling Language
URI	Uniform Resource Identifier
YAFFS2	Yet Another Flash File System

1. INTRODUCTION

In recent years, powerful smartphones and tablet devices are popular choice for consumers. The device manufacturers are developing more and more powerful, stylish and versatile mobile devices with more memory and processing power as well as hardware features like GPS, accelerometers, and touchscreens. This trend is providing mobile developers a big opportunity to create innovative mobile applications.

Currently, mobile devices market offers several competing mobile phone software stacks like Symbian from Nokia, Windows Mobile from Microsoft, iPhone OS from Apple, Blackberry OS from RIM, LiMo (Linux Mobile) and Android from Google. Many of these existing platform use proprietary, relatively closed software stacks, such as Nokia's Series 60 with the Symbian operating system, or Microsoft's Windows Mobile. Modifications to these stacks have to be done either by the stack owner or by the handset manufacturer. The stacks are not open source, so changing anything in the stack is difficult at best. Most Linux-based phones to date have an open source kernel, but keep other details of the software stack (application framework, multimedia framework, applications) proprietary.

Some studies [1] predict that the open-source platforms will continue to dominate the smartphone market and Android is an increasing factor in smartphone computing. Android introduces new extensions and features to the Linux kernel. Many proprietary operating systems restrict the development and deployment of third-party applications. On the other hand, Android includes an open catalog of applications, Android Market (Google Play), that users can download over the air to their Android phones. Android developers are free to develop creative applications that take full advantage of increasingly powerful mobile hardware and distribute them in an open market.

One of the main intentions of Android platform is to eliminate the duplication of functionality in different applications to allow functionality to be discovered and invoked on the fly, and to let users replace applications with others that offer similar functionality. The main problem here is how to develop applications that must have as few dependencies as possible, and must be able to provide services to other applications.

This thesis studies the Android system, its capabilities in developing applications that communicate with each other and provide services to other applications. A sample application called “EventPlanner” has been developed using the client and server architecture. The client component implements user interface of the application and Server component implements SMS communication, database, and processes the client’s requests. The Client and Server communicate using Inter Process Communication (IPC). This application uses Android Interface Definition Language (AIDL) to implement and

use remote methods. This work also gives an overview of Android's Intent mechanism and Binder component which are different levels of abstraction of IPC mechanisms in Android.

This thesis is structured as follows. Chapter 2 presents the Android platform, its features, architecture, and application components. Chapter 3 explains the Android processes, and threads. Chapter 4 explains the use of AIDL to implement Inter Process Communication in Android. Additionally, it introduces the Android's Intent mechanism and Binder component. Chapter 5 discusses the development of sample application including its architecture, design and implementation. Chapter 6 provides evaluation of the platform and IPC. Chapter 7 presents conclusions of the thesis.

Throughout this thesis, it is supposed that the reader knows the basics of mobile development technologies, constraints, and the basics of operating systems. Understanding the development of sample application is made easier by knowing the basics in object oriented design.

2. THE ANDROID PLATFORM

This chapter describes what Android is and how it works. This chapter also explains the features of Android, its architecture, the application components.

2.1. What is Android?

Android is a free open source software platform for mobile devices. The project is led by Google and developed by Open Handset Alliance (OHA), a group of technology and mobile companies [2], founded in November 2007.

Google's Andy Rubin describes Android as follows: "Android is the first truly open and comprehensive platform for mobile devices. It includes all of the software (operating system, user-interface and applications etc.) to run a mobile phone, but without the proprietary obstacles that have hindered mobile innovation." [3]

The first beta version of the Android Software Development Kit (SDK) was released in November of 2007 [4] and non-beta version (v1.0) was made public in September 2008. Platform has seen a number of updates to its base operating system since its original release. These updates typically fix bugs and add new feature. The latest version of Android is 4.0.3 (API Level 15), released in December 2011. *Figures 2.1* and *2.2* show sample Android phones.

Figure 2.1. Android Emulator (v 1.5)

Figure 2.2. Galaxy Nexus Android 4.0.1

2.2. Android features

Android platform has the following features [5]:

- **Open source.** There are no licensing, distribution, or development fees to use and customize Android.
- **Java Programming.** Android applications are written in Java, run in Dalvik Virtual Machine (DVM) which is optimized for low memory requirements.

- **Access to hardware.** SDK includes APIs to access hardware such as GPS, camera, audio, network connections, Wi-Fi, Bluetooth, accelerometers, the touchscreen, and power management.
- **Background Services.** Android supports applications and services designed to run invisibly in the background. Background services make it possible to create invisible application components that perform automatic processing without direct user action.
- **Optimized memory and Process management.** Android uses its own run-time and virtual machine to manage application memory. Android run-time also manages the process lifetimes. Android ensures application responsiveness by stopping and killing processes as necessary to free resources for higher-priority applications.
- **Shared data and Inter application communication.** Android includes techniques for easily transmitting information from one application to other applications.
- **Maps and Location based services.** Android APIs allow developers to create wide range of map-based applications using GPS and cell based technologies.
- **Media support.** Android provides graphics libraries for 2D canvas drawing and 3D graphics with OpenGL. Android also offers comprehensive libraries for handling still images, video, and audio files, including the MPEG4, H.264, MP3, AAC, AMR, JPG, PNG, and GIF formats.

2.3. Android architecture

Android is based on Linux kernel, a collection of C/C++ libraries, and application framework that provides services for the applications and manages the run-time [5]. *Figure 2.3* shows the major components of the Android operating system.

Android relies on Linux Kernel 2.6 to handle the core system services such as security, memory management, process management, network stack, and driver model. The kernel also acts as an abstraction layer between the hardware and the rest of the software stack. Kernel does not include some of standard Linux utilities such as native windowing system, glibc and Google has added some enhancements to the kernel to support various aspects of Android that differ from regular Linux.

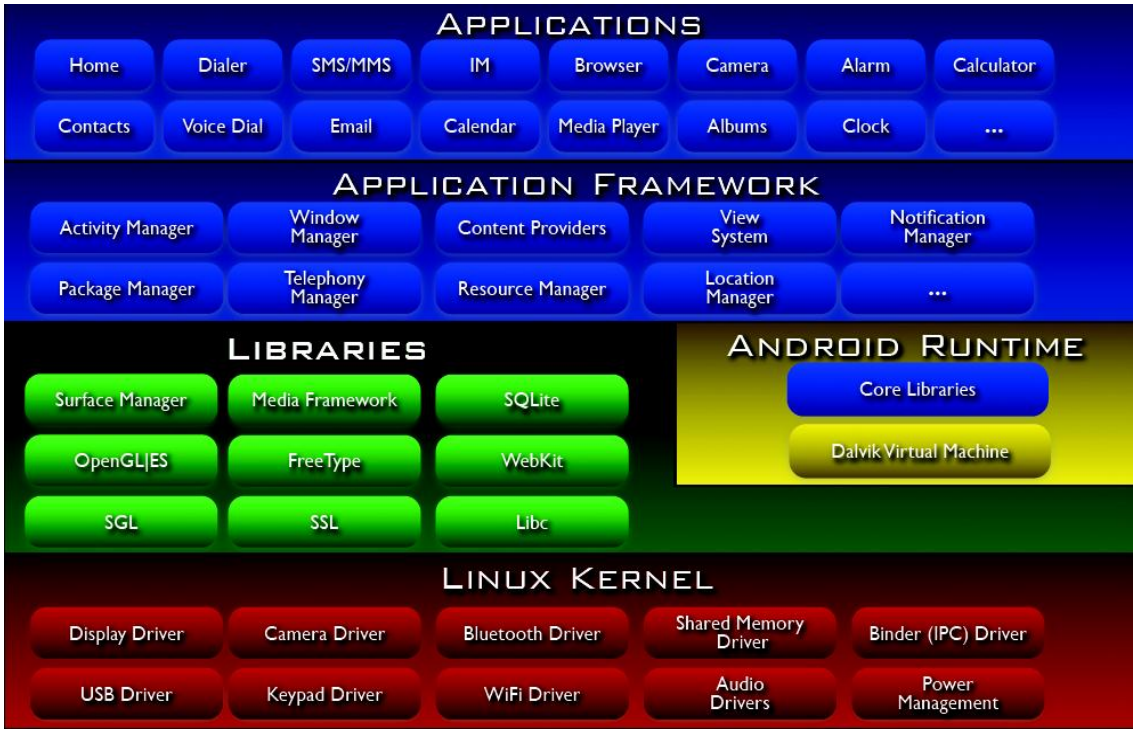

Figure 2.3. Android architecture [6]

The following kernel features are unique to Android [7, 8]:

- **Binder -** Android specific inter process communication mechanism, and remote method invocation system. Binder is explained in detail in later chapter.
- **Logger -** System logging facility. System wide logging is supported by a kernel device driver called ‘logger’. The code for the driver is in ‘drivers/android/logger.c’. The code supports 4 logging buffers, named "main", "events", "radio", and "system" for different types of events. [9]
- **Wakelocks -** Used for power management. It prevents the system to go into low power state. Wakelocks implementation is available in ‘kernel/power/wakelock.c’. [10]
- **Early suspend -** Early suspend is an extension to the standard Linux power management stages. This does not suspend the device, but some of its components like LCD, touch sensor and accelerometer. The kernel calls early suspend handlers when the user requested sleep state changes.
- **Out Of Memory Handler -** OOM handler automatically kills less important processes as available memory becomes low. Implementation is available at ‘drivers/misc/lowmemorykiller.c, security/lowmem.c’.
- **Ashmem -** The Ashmem subsystem is a new shared memory allocator similar to POSIX SHM but with different behaviour and providing a simpler file based

API. It better supports low memory devices, because it can discard shared memory units under memory pressure.

- **Android Timed output -** Timed output is a system to allow changing a gpio pin and restore it automatically after a specified timeout. This exposes a user space interface for timed GPIOs. It is used in the vibrator code.
- **YAFFS2 -** Flash memory file-system. YAFFS2 was already freely available for Linux. However, it is not part of the standard Linux kernel, and so Google added it to Android.

Android Runtime includes a set of core libraries that provides most of the functionality available in the core libraries of the Java programming language. Every Android application runs in its own process, with its own instance of the Dalvik virtual machine. The Dalvik VM executes files in the Dalvik Executable (.dex) format, which is optimized for minimal memory footprint. The DVM is register-based, and runs classes compiled by a Java language compiler that have been transformed into the .dex format by the included "dx" tool. The Dalvik VM relies on the Linux kernel for underlying functionality such as threading and low-level memory management. [11, 12]

The **Libraries**, running on top of the kernel, a set of various C/C++ core libraries exposed through an application framework to the application layer. Following are some of the core libraries [11]:

- **System C library -** BSD-derived implementation of the standard C system library (libc), optimized for embedded Linux-based devices.
- **Media Libraries** - Based on PacketVideo's OpenCORE. The libraries support playback and recording of many audio and video formats, as well as static image files, including MPEG4, H.264, MP3, AAC, AMR, JPG, and PNG.
- **Surface Manager** - Manages access to the display subsystem and seamlessly composites 2D and 3D graphic layers from multiple applications.
- **WebKit** - Web browser engine which provides tools for browsing the web.
- **SGL** - The underlying 2D graphics engine.
- **3D libraries** - An implementation based on OpenGL ES 1.0 APIs. These libraries use either hardware 3D acceleration or the highly optimized 3D software rasterizer.
- **FreeType** - Bitmap and vector font rendering.
- **SQLite** - Lightweight relational database engine available to all applications.

Application Framework provides the classes used to create Android applications. It also provides a generic abstraction for hardware access and manages the user interface and application resources. The application framework contains the following components:

- **Activity Manager** – Controls life cycle of activities and manages Activity stack.

- **View System** - Used to construct user interfaces for the activities.
- **Content Providers** - Manages access to a central repository of data and allows applications to share data.
- **Resource Manager** - Provides access to non-code resources like localized strings, graphics, and layout files.
- **Window Manager** - Java programming language abstraction on top of lower services. It manages windows using the services of surface manager.
- **Package Manager** - Manages the applications that are installed on the device.
- **Notification Manager** - Enables applications to display custom alerts in the status bar.
- **Telephony Manager** - Contains APIs to build phone applications.

Application layer is a set of core applications, both native (contacts, SMS program, calendar, browser etc.) and third-party that are built on the application layer by means of the same API libraries. The application layer runs within the Android run-time, using the classes and services made available from the application framework.

To develop the whole Android system, four programming languages are used: Assembler, C, C++ and Java. The kernel has a small amount of Assembler but is mainly written in C. Some native applications and libraries are written in C++. All other applications, especially custom applications are written in Java. Performance-critical portions of the applications can be written in C++.

2.4. Android application basics

Android applications are written in Java programming language. SDK tools compile the source codes along with any data and resource files into an archive file with an .apk suffix. This is called an Android package and all the code in single .apk file is treated as one application [13]. When an Android package is installed on Android powered device, each application lives in its own security sandbox which means the Android OS allows any application/process to be able to access only the information and resources that are necessary for its legitimate purpose. This is known as principle of least privilege and is achieved through Android platform principles listed as follows:

- The Android operating system is a multi-user Linux system in which each application is a different user.
- Each application is assigned a unique Linux user ID which will be used only by the system and is unknown to the application. The system sets permissions for all the files in an application so that only the user ID assigned to that application can access them.
- Each process has its own virtual machine (VM), so an application's code runs in isolation from other applications.

- By default, every application runs in its own Linux process. Android starts the process when any of the application's components need to be executed, then shuts down the process when it is no longer needed or when the system must recover memory for other applications.

The above principles make the Android platform a very secure environment in which an application cannot access parts of the system for which it is not given permission. However, according to [13], there are ways for an application to share data with other applications and for an application to access system services:

- It is possible to arrange for two applications to share the same Linux user ID, in which case they are able to access each other's files. To conserve system resources, applications with the same user ID can also arrange to run in the same Linux process and share the same VM (the applications must also be signed with the same certificate).
- An application can request permission to access device data such as the user's contacts, SMS messages, the mountable storage (SD card), camera, Bluetooth, and more. All application permissions must be granted by the user at install time.

2.5. Application components

Android applications consist of loosely coupled components that are essential building blocks of an Android application. These components are bounded by an application manifest that describes each component and how they all interact, as well as the application meta-data. Android's architecture encourages the concept of component reuse by enabling the application to publish and share application components and data with other applications with provided access rights. [5]

Each application component is a different entry point of the application. However, not all components are actual entry points for the user and some depend on each other, each one exists as its own entity and plays a specific role that helps to define the application's overall behaviour. There are four different types of application components namely Activities, Services, Content providers, Broadcast receivers. Each type offers a distinct purpose and has a distinct life-cycle that defines how the component is created and destroyed [13, 14]. Android application is collection of one or more of these application components as shown in *Figure 2.4*.

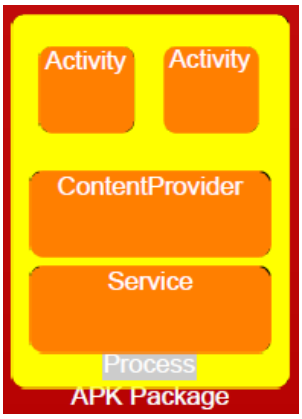

Figure 2.4. Android package [14]

2.5.1. Activities

An Activity [15] represents a single screen with a user interface. Every screen in the application will be an extension of the Activity class. Activities use Views to form graphical user interfaces that display information and respond to user actions. Usually most Activities are designed to occupy the entire display, but it is possible to create semitransparent or floating activities. An activity can be primarily in three states, mentioned in the following.

- **Active** or **Running** - When the activity is in the foreground of the screen and user is interacting with activity.
- **Paused** - When activity is visible but does not have focus. That is when another activity which may be transparent or non-full-screen is on top of Activity Stack and has focus. So, the paused activity does not receive user input events.
- **Stopped** - When an Activity is not visible, it is stopped by the framework. The Activity remains in memory retaining all state information. But when the memory is very low and the system requires memory for some other important tasks, Stopped activity will be terminated. When an Activity is stopped it is important to save data and the current UI state. After an Activity has been killed, and before it has been launched, it becomes inactive.

State transitions are sometimes non-deterministic, and Android memory manager is completely responsible to handle it [5]. When an activity transitions into and out of the different states described above, it is notified through various callback methods. All of these callback methods can be overridden to do necessary work when the state of the activity changes. *Figure 2.5* illustrates transitions an activity might take between states.

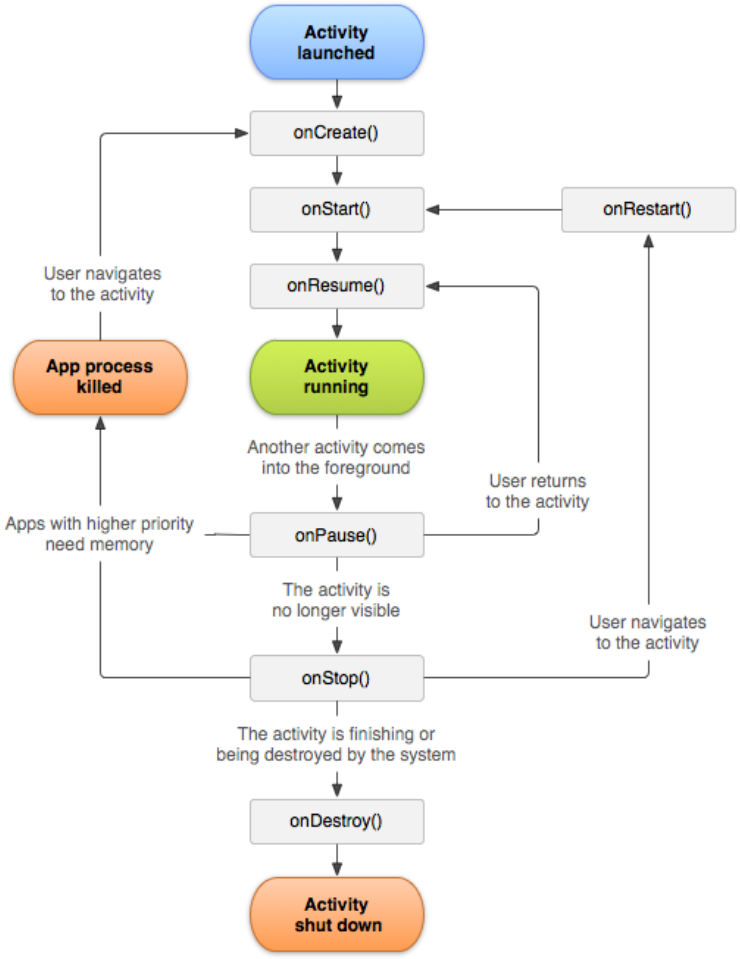

Figure 2.5. Activity Life cycle [15]

Following list describes the each lifecycle callback methods shown in *Figure 2.5*.

- `onCreate()`: This method is called when the activity is first created. This method must be overridden and all initializations such as variable initializations, creating views, binding data to lists, etc. should be done in this method.
- `onStart()`: This method is called just before the activity becomes visible to the user. This is followed by `onResume()` if the activity comes to the foreground, or `onStop()` if it becomes hidden.
- `onRestart()`: This method is called after the activity has been stopped, just prior to it being started again. Always followed by `onStart()`.
- `onResume()`: This method is called just before the activity starts interacting with the user. At this point the activity is at the top of the activity stack, with user input going to it. Always followed by `onPause()`.
- `onPause()`: This method is called when the system is about to start resuming another activity. This method is typically used to commit unsaved changes to persistent data, stop animations and other things that may be consuming CPU,

etc. It should do whatever it does very quickly, because the next activity will not be resumed until it returns.

- `onStop()`: This method is called when the activity is no longer visible to the user. This may happen because it is being destroyed, or because another activity has been resumed and is overlapping it.
- `onDestroy()`: This method is called before the activity is destroyed. This is the final call that the activity will receive. It could be called either because the activity is finishing, or because the system is temporarily destroying this instance of the activity to save space.

Since Activity often changes its state during its run-time, it is important to have good understanding of the Activity life cycle to ensure that the application provides a seamless user experience and properly manages its resources. In this thesis work Activity components are used extensively in combination with Service component to study IPC mechanism provided by Android platform.

2.5.2. Services

Services [16] perform long-running operations or work for remote processes in the background and do not provide a user interface. Other component, such as an activity, can start a service and it will continue to run in the background even if the user switches to another application. Additionally, a component can bind to a service to interact with it and even perform inter process communication. For example, a service might handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.

Services are started, stopped, and controlled from other application components, including other Services, Activities, and Broadcast Receivers. Started Services always have higher priority than inactive or invisible Activities, making them less likely to be terminated by the run-time's resource management. A service can essentially take two forms:

- **Started Service** An application component such as an activity starts the service by calling `startService()`. Once the Service is started, it can run in the background indefinitely, even if the component that started it is destroyed. Started service usually performs a single operation like downloading a file over the network, and does not return a result to the caller. Service is stopped when the operation is done.
- **Bound Service** An application component binds to Service by calling `bindService()`. A bound service provides client-server interface that allows components to interact with the service. Application components can send requests, get results, and perform inter process communication. A bound service runs only as long as another application component is bound to it. When multiple components are bound to the service at once, Service is destroyed when all of them are unbound.

Service is created by creating a subclass of `Service`. Service implementation needs to override some callback methods to monitor changes in service's lifecycle and provide a mechanism for components to bind to the service. *Figure 2.6* shows the service life cycle. The left part shows the lifecycle when the service is created with `startService()` and the right part shows the lifecycle when the service is created with `bindService()`.

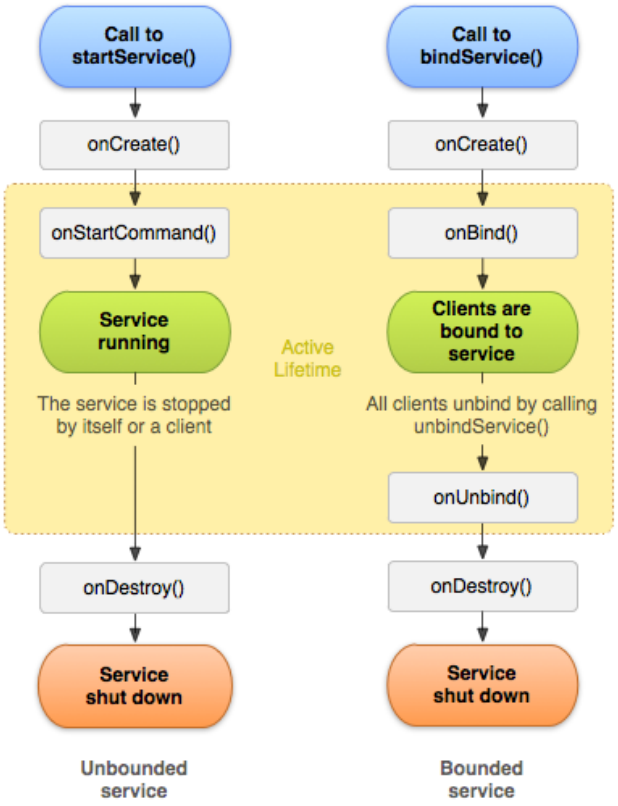

Figure 2.6. Service Lifecycle [16]

The **entire lifetime** of a service happens between the time `onCreate()` is called and the time `onDestroy()` returns. Service performs its initial setup in `onCreate()` and releases all remaining resources in `onDestroy()`. The `onCreate()` and `onDestroy()` methods are called for all services, whether they are created by `startService()` or `bindService()`.

The **active lifetime** of a service begins with a call to either `onStartCommand()` or `onBind()`. Each method is handed the Intent that was passed to either `startService()` or `bindService()`, respectively. If the service is started, the active lifetime ends the same time that the entire lifetime ends. If the service is bound, the active lifetime ends when `onUnbind()` returns.

2.5.3. Content providers

Content provider [17] manages a shared set of application data and provides a generic way to share information between the applications. For example, contact information is shared through a content provider to any application. A content provider is often accessed from another process by means of IPC. *Figure 2.7* shows how a Content provider exists within Android system.

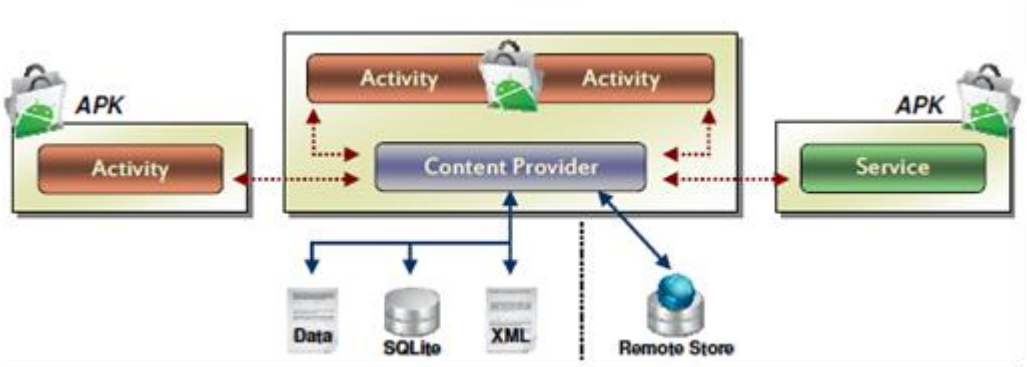

Figure 2.7. Content provider [18]

2.5.4. Broadcast receivers

A Broadcast Receiver [19] is a component that responds to system-wide broadcast announcements such as incoming call, SMS received, screen has turned off, the battery is low, or a picture was captured. Applications can also initiate broadcasts, for example, to inform other applications that some data has been downloaded to the device and is available for them to use. *Figure 2.8* shows how a Broadcast receiver exists within Android system.

Figure 2.8. Broadcast receiver [18]

When designing an application, by decoupling the dependencies between application components, it is possible to share and interchange individual components with other applications. Also, Android application must specify all its components in AndroidManifest.xml file so that Android system can start a component [20].

3. PROCESS AND THREADS

This chapter discusses how processes and threads work in Android application, process states and lifecycle.

3.1. Process

As already mentioned, by default every Android application is executed in a new Linux process. Each process is started with a single thread of execution when an application component is started and the application does not have any other components running at that moment. All components of the same application run in the same process and thread, called the "main" thread.

When an application component is started and there is already a process exists for that application, then the component is started with in that process and uses the same thread of execution. However, it is possible to force different components within the same application to run in separate processes or to have multiple applications share the same process using the `android:process` attribute on the affected component nodes within the manifest. In addition, each application runs in a separate instance of the Dalvik Virtual Machine. [21]

3.2. Process lifecycle

Android applications have limited control over their own life cycles. Alternatively, Application components must listen for changes in the application state and react accordingly, taking particular care to be prepared for unexpected termination. Android aggressively manages its resources, doing whatever it takes to ensure that the device remains responsive. This means that processes (and their hosted applications) will be killed, without warning in some cases, to reclaim resources for new or more important processes - generally those interacting directly with the user at the time. [5]

To determine the order in which processes are killed to reclaim resources, the system places each process into an "importance hierarchy" based on the components running in the process and the state of those components. Application's priority is equal to its highest-priority component. When there are more applications with same priority, the process that has been at a lower priority longest will be killed first. Process priority is also affected by inter process dependencies; if an application has a dependency on a Service or Content Provider supplied by a second application, the secondary application will have at least as high a priority as the application it supports. All Android applica-

tions will remain running and in memory until the system needs its resources for other applications. [21]

It is essential to understand the priority of processes so that the application can be structured correctly to make sure that its priority is appropriate for the task it is doing. Otherwise, the application could be terminated while it is in the middle of something important. *Figure 3.1* shows the importance hierarchy used to determine the order of process termination.

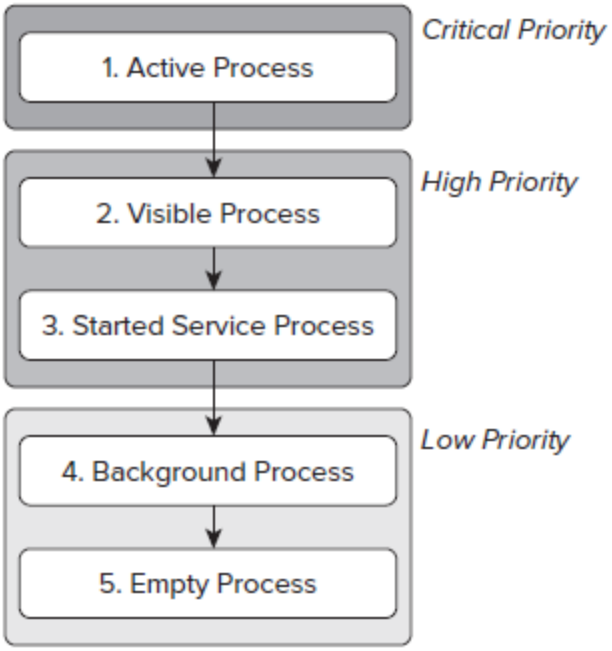

Figure 3.1. Process priority [5]

The following list presents each of the process states shown in *Figure 3.1*, explaining how the state is determined by the application components comprising it. [5, 21]

Active or **Foreground Processes** are those hosting applications with components currently interacting with the user. These are the critical priority processes Android tries to keep responsive by reclaiming resources. There are usually very few of these processes at a time, and they will be killed only as a last resort, when the memory is critically low. Active processes comprise the following categories.

- Activities that are in "active" state, which means they are in the foreground and responding to user events.
- Activities, Services, or Broadcast Receivers that are currently executing an `onReceive` event handler.
- Services that are executing `onStart`, `onCreate`, or `onDestroy` event handlers.
- Running Services that have been flagged to run in the foreground.

Visible Processes are those that are visible but inactive. These processes are not in foreground or interacting with user events. This happens when an Activity is partially obscured (by a non-full-screen or transparent Activity). There are generally very few

visible processes, and they will only be killed in extreme circumstances to allow active processes to continue.

Started Service processes are those hosting Services that have been started. These processes receive a slightly lower priority than visible Activities because they do not interact with the user directly but should continue on-going processing without a visible interface. They are still considered as foreground processes and will not be killed unless resources are needed for active or visible processes.

Background Processes are those hosting Activities that are not visible and that do not have any Services that have been started. There would usually be a large number of background processes that Android will kill using a LRU (Least Recently Used) policy to reclaim resources for foreground processes.

Empty Processes are those that are retained in memory after they have reached the end of their lifetimes. Android maintains this cache to improve overall system performance by reducing start-up time of applications when they are re-launched. These are the lowest priority tasks and are killed as required.

3.3. Threads

A Thread is a concurrent unit of execution. Thread has its own call stack (execution stack) to store information about the methods (subroutines) being invoked, their arguments and local variables [22]. Multi-Threading concept is important in most platforms. It provides maximum utilization of the processor. Threading is used when the program executes time consuming processes such as a web service and to give a good user experience by not blocking the UI.

Android API provides standard Java Thread object to support threading. When an Android application is launched, the system creates a thread of execution for the application, called "main". This thread is responsible for dispatching drawing events and user events to the appropriate user interface widgets. Also, Android applications interact with Android UI toolkit components (components from the `android.widget` and `android.view` java packages) from this main thread. This thread is also called as UI thread. All components that run in the same process are instantiated in the UI thread, and system calls to each component are dispatched from that thread. Consequently, methods that respond to system callbacks such as `onKeyDown()` to report user actions or a lifecycle callback method always run in the UI thread of the process. [21]

According to [21, 23], when the application (any of app components) is performing time-consuming, intensive work in the UI thread, it will block the whole UI. When the thread is blocked, no events can be dispatched, including drawing events and the application appears to be hanged from the user's perspective. In Android, Activities that do not respond to an input event (such as a key press) within five seconds, and Broadcast Receivers that do not complete their `onReceive()` handlers within 10 seconds, are considered unresponsive and the user is presented with the "application not responding" (ANR) dialog. The user can then decide to quit the application.

So, to ensure that the applications remain responsive, it is good practice to move all slow and time-consuming operations including file operations, network lookups, database transactions, and complex calculations from UI thread onto background or worker threads [5]. In addition, the Android UI toolkit is not thread-safe. So, all manipulations to UI must be performed from UI thread but not from a worker thread. Consequently, two main rules of Android's thread model are (1) UI thread should not be blocked and (2) Android UI toolkit should not be accessed from outside the UI thread [21]. Android offers AsyncTasks and Handlers for threading, which are described in the following sub sections.

3.3.1. AsyncTask

AsyncTask [24] allows performing long running tasks in the background, and then provides event handlers to monitor and publishes the results on the UI thread. Asynchronous task is implemented by creating a class that extends `AsyncTask` and implementing the different protected methods it provides.

Creating a new Asynchronous Task:

Implementation should specify the classes (data types) used for input parameters on the execute method, the progress-reporting values, and the result values in the following format:

```
AsyncTask<[Input Parameter Type], [Progress Report Type],
[Result Type]>
```

- Input Parameter Type: The type of the parameters sent to the task upon execution
- Progress Report Type: The type of the progress units published during the background computation
- Result Type: The type of the result of the background computation

If there is no need to take input parameters, update progress, or report a final result, just specifying void for any or all of the types is required.

Example:

```
private class TestAsyncTask extends AsyncTask<String, Inte-
ger, Integer> {

}
```

When an asynchronous task is executed, the task goes through 4 steps and subclass of AsyncTask should implement the following event handlers:

- `onPreExecute()` is invoked on the UI thread immediately after the task is executed. This step can be used to setup the task, for instance a UI indication like a progress bar can be shown to indicate the long running operation.
- `doInBackground(Params...)` is invoked on the background thread immediately after `onPreExecute()` finishes executing. All the time consuming computation is performed in this step. The parameters of the asynchronous task are passed to this step. The result of the computation must be returned by this step and will be passed back to the last step. This step can also use `publishProgress(Progress...)` to publish one or more units of progress. These values are published on the UI thread, in the `onProgressUpdate(Progress...)` step.
- `onProgressUpdate(Progress...)` is invoked on the UI thread after a call to `publishProgress(Progress...)`. This method is used to display any form of progress in the user interface while the background computation is still on-going. Execution time is undefined for this method.
- `onPostExecute(Result)` is invoked on the UI thread after the background computation finishes. The result of the background computation is passed to this step as a parameter.

Running an Asynchronous Task:

Asynchronous task is executed by creating a new instance and calling `execute()` method with parameters, each of the type specified in the implementation.
Example:

```
new TestAsyncTask().execute("inputString1","inputString2");
```

Each AsyncTask instance can be executed only once. When attempted to call `execute()` a second time, an exception will be thrown. Also, a task can be cancelled at any time by invoking `cancel(Boolean)`.

3.3.2. Using Handlers

Sometimes it is necessary to create and manage threads manually to perform background processing. This section explains how to create and start new Thread objects, and how to synchronize with the UI thread using Handlers.

Android provides `java.lang.Thread` and `android.os.Handler` classes to create and manage child threads. According to [25], Handler allows to send and process `Message` and `Runnable` objects associated with a thread's message queue. Each `Handler` instance is associated with a single thread and that thread's message queue. When a new `Handler` is created, it is bound to the thread / message queue of the thread that is creating it. Once the `Handler` is bound to the message queue of the thread, it will deliver messages and runnables to that message queue and execute them

as they come out of the message queue. Two main uses of a Handler are (1) it schedules messages and runnables to be executed at some point in the future and (2) to enqueue an action to be performed on a different thread than your own.

There are two main ways of having a Thread execute application code. One is providing a new class that extends Thread and overriding its `run()` method. The other is providing a new Thread instance with a `Runnable` object during its creation. In both cases, the `start()` method must be called to actually execute the new Thread. Each Thread has an integer priority that determines the amount of CPU time available to the Thread. The method `setPriority(int)` is used to set the priority. A Thread can also be made a daemon, which makes it run in the background. [22]

Thread creation syntax:

```
Thread (ThreadGroup group, Runnable runnable, String threadName, long stackSize)
```

Example: Creating and running a new thread

```
// This is called on the main UI thread
Thread thread = new Thread(null, doLongRunningProcessing, "Background");
thread.start();
```

Here is the Runnable object that executes the long running operations.

```
Private Handler handler = new Handler();
private Runnable doLongRunningProcessing = new Runnable() {
   public void run() {
    // perform time consuming operations…
   }
};
```

Here is how to use Handler class in UI thread, to process messages from background threads.

```
Handler handler = new Handler() {
  @Override
  public void handleMessage(Message msg) {
   // update widgets and any other UI elements
  }
 };
```

To summarize this, when a process is started, its main thread is dedicated to running a message queue that takes care of managing the top-level application objects such as activities, broadcast receivers, and any UI elements they create. New threads can be

created and communicate back with the main application thread through a `Handler` by calling the `post` or `sendMessage` methods from new threads. The given `Runnable` or `Message` will then be scheduled in the Handler's message queue and processed when appropriate. [25]

4. INTER PROCESS COMMUNICATION

This chapter introduces different level of Inter Process Communication (IPC) mechanisms in Android and explains the implementation of remote methods using Android Interface Definition Language (AIDL).

4.1. Background

IPC is a set of methods for exchanging data among multiple threads in one or more processes. IPC methods are divided into methods for message passing, synchronization, shared memory, and remote procedure calls (RPC). The method of IPC used may depend on the bandwidth and latency of communication between the threads, and the type of data being communicated. Some of the reasons for providing an environment that allows process communication include information sharing, speedup, modularity and security.

One of the important features of Android platform is that tries to eliminate the duplication of functionality in different applications. Functionality can be discovered and invoked on the fly, and users can replace applications with others that offer similar functionality. Applications must have as few dependencies as possible, and must be able to contract out operations to other applications that may change at the user's choice. Consequently, IPC is the basis of key features of the Android programming model. *Figure 4.1* shows the abstraction of IPC mechanism in Android platform.

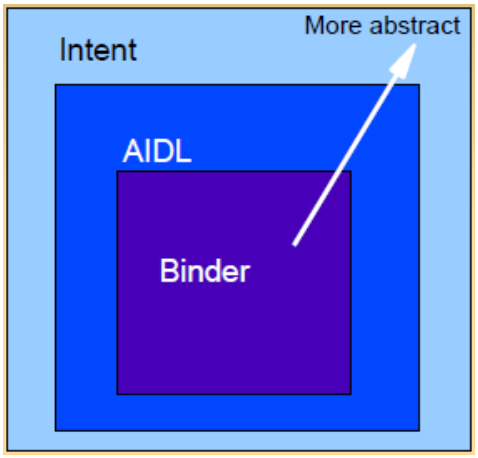

Figure 4.1. Abstraction of IPC in Android [26]

Intent provides a simple and high-level system of inter process communication. It enables applications to select an Activity or Service based on the action user wants to invoke and the data on which they operate. In other words, it does not need a hardcoded path to an application to use its functions and exchange data with it. Intent objects also deliver data from one application to another.

AIDL resembles the remote procedure calls offered by other systems. AIDL is used to create rich application interfaces that support full object-based inter process communication between applications running in different processes. It makes APIs accessible remotely. Remote objects allow you to make method calls that look "local" but are executed in another process. This IPC technique is the main focus of this thesis work and is explained in detail in the following sections.

Binder is a kernel driver that facilitates inter process communication mechanism, and remote method invocation. This is lowest level abstraction of IPC in Android and offers high performance through shared memory.

4.2. Remote methods and AIDL

AIDL provides Service component the ability to support multiple applications across process boundaries and it allows to define the programming interface that both the client and service agree upon in order to communicate with each other using inter process communication. In Android, as mentioned already, one process cannot normally access the memory of another process. So to pass objects/data between processes, it is needed to decompose the objects into OS-level primitives that the underlying operating system can understand. That means the data has to be "marshalled", packaged for transport and "unmarshalled", put into the right member variables after the data has been moved across the process boundary. Marshaling objects is tedious process but, AIDL simplifies the marshalling and exchange of objects [4]. Following are the steps to create and use remote methods using AIDL:

1. Define the interface in the AIDL
2. Implement the interface
3. Expose the interface to clients
4. Invoke the remote methods from clients

4.2.1. Creating AIDL interface definition

According to [27], AIDL interface is defined in an .aidl file using the Java programming language syntax. This file should be saved in source code of both the application hosting the service and client applications that need to bind to the service. Only one interface definition is allowed in one .aidl file and requires only the interface declaration and method signatures. AIDL supports the following data types:

- All primitive types in Java programming language such as int, boolean, float, char, etc.

- String and CharSequence values.
- List (including generic) objects, where each element is a supported type. The receiving class will always receive the List object instantiated as an ArrayList.
- Map (not including generic) objects, when every key and element is of a supported type. The receiving class will always receive the Map object instantiated as a HashMap.
- Classes that implement the Parcelable. This needs import statement.
- An `import` statement must be included for each additional type not listed above, even if they are defined in the same package as interface.

Methods in the interface can take zero or more parameters and return a value or void. All non-primitive parameters require a directional to indicate if the parameter is a value or reference type using the `in`, `out`, and `inout` keywords. Any parameter labelled `in` will be transferred to the remote method, whereas any parameter labelled `out` will be returned to the caller from the remote method. And any parameter labelled `inout` will transfer data to the remote method and refer to a value transferred from the remote method when it returns. Primitives are in by default, and cannot be otherwise. Since marshalling parameters is expensive operation, it is good practice to limit the direction of each parameter.

Following example code is excerpted from the IEventPlannerService.aidl file in the EventPlanner application. This is how we specify an interface to a remote object:

```
interface IEventPlannerService {
   void sendNewEvent(
         String phoneNumberList,
         String message);

   void sendEventReply(
         String phoneNumber,
         String eventReplySMS);

   void sendEventConfirmation(
         String phoneNumberList,
         String eventConfirmationSMS);

   String retriveNewEvent();
}
```

There are method signatures, and no implementation of the methods. That is all AIDL needs to create code that moves the parameters between applications. When the AIDL file is saved in Eclipse, the Android Eclipse plug-in compiles it and generates the IBinder interface file in the project's gen/ directory. The generated file name matches

the .aidl file name, but with a .java extension (for example, IEventPlannerService.aidl results in IEventPlannerService.java). Both the calling and implementing side of a remote method interface share the information in the AIDL file. The java code (IEventPlannerService.java) generated by AIDL is given in APPENDIX 2.

4.2.2. Implementing AIDL interface

As described in previous section, when the AIDL file is saved in Eclipse, the Android build tools generates java file. Below is the short version of IEventPlannerService.java generated by AIDL.

```
public interface IEventPlannerService extends
adroid.os.IInterface
{
    /** IPC implementation stub class. */
    public static abstract class Stub extends
    adroid.os.Binder implements
    com.ipc.eventplannerservice.IEventPlannerService
    {
    …
    …
        private static class Proxy implements
        com.ipc.eventplannerservice.IEventPlannerService
        {

        } // end of Proxy

    } // end of stub

} // end of IEventPlannerService
```

The class `android.os.IInterface` is a base type on which all the interfaces created by AIDL are built. `IEventPlannerService` extends `android.os.IInterface`. Most of the code in the `IEventPlannerService` interface is part of the definition of an abstract class called Stub. And remote methods are implemented by extending the Stub class. Every remote interface has this class. The word "stub" was chosen to refer to this class because remote method systems work by creating a method on the client with the same name as the method that runs on the server. The client method is considered a "stub" because it does not actually carry out the operation requested. It just marshals the data, sends it to the server, and unmarshalls the return value. [4, 27]

To actually implement these remote methods, extend the generated `Stub` class (`IEventPlannerService.Stub`) which is extended from `Binder` interface, and implement the methods inherited from the .aidl file. The following code snippet, taken from the EventPlannerService.java file in the EventPlanner application shows the method definitions.

```
public class EventPlannerServiceImpl extends
IEventPlannerService.Stub
{
    @Override
    public void sendNewEvent(
          String phoneNumbers,
          String message) throws RemoteException {

          // implementation
    }

    @Override
    public void sendEventReply(
          String phoneNumber,
          String eventReplySMS) throws RemoteException {

          // implementation
    }

    @Override
    public void sendEventConfirmation(
          String phoneNumberList,
          String eventConfirmationSMS)
          throws RemoteException {

    // implementation
    }
}
```

This is all that is needed to do to turn a method in the application into a remote method. The rest of the work of invoking the method in the other application, passing the parameters, and responding with a return value from the remote method is performed by code generated by AIDL in the Stub abstract class. *Figure 4.2* shows the UML representation of AIDL generated code and how client (caller) and service (callee) use AIDL generated code.

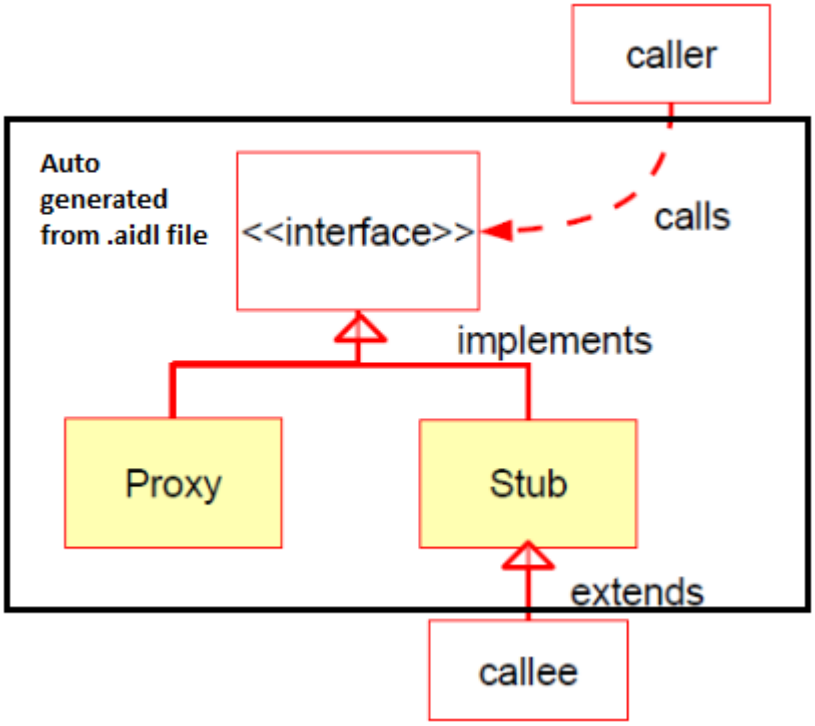

Figure 4.2. UML representation of AIDL generated code

Following are some of the important rules to be aware of when implementing AIDL interface:

- All exceptions will remain local to the implementing process. That means exceptions will not be sent back to the calling application.
- By default, RPC calls are synchronous. If the Service process is likely to take more than few milliseconds to perform a request, IPC method should not be called from the activity's main thread to avoid blocking of UI. Instead, it should be called from a separate thread in the client.

4.2.3. Exposing the interface to the clients

Once all the methods in the interface are implemented, server (Service component) needs to publish the interface to client applications so they can bind to it. Publishing is done by overriding the `onBind()` method of the Service class. The method `onBind()` should return an instance of the class that implements the AIDL generated Stub. Following code snippet is taken from EventPlanner application that shows how to publish the IEventPlannerService interface to clients. Below is the onBind implementation in service side.

```
public class EventPlannerService extends Service {
…
        @Override
        public IBinder onBind(Intent intent) {
```

```
                    // return the interface
                    return new EventPlannerServiceImpl();
            }
…
}
```

Implementation of class `EventPlannerServiceImpl` is shown in previous section. Below code snippet shows client side code to bind to a service.

```
private ServiceConnection serviceConn;
bindService(
            new Intent(IEventPlannerService.class.getName()),
            serviceConn,
            Context.BIND_AUTO_CREATE);
```

A client calls the `bindService()` method of the Context class, causing a call to the server's onBind method. The bindService and onBind methods are the "handshake" required to start using a remote interface in a specific Service object in a specific process running in the Android environment.

In the client application looking for this interface, the contents of the Intent object were specified in a call to the `bindService` method of the `Context` class. That means that a program publishing a remote method interface must be a subclass of `Service`. But a program using a remote method interface can be any subclass of `Context`, including `Activity` and `Service`. The `Intent` object is used to specify the interface. The class name of the interface is the `action` parameter of the `Intent`. If the interface matches, the `onBind` method returns an IBinder instance, an instance of the Stub interface in the remote interface. [4]

4.2.4. Invoking remote methods from the clients

This section describes `asInterface()` method of the Stub class, and the Proxy class nested within Stub, which are important to allow the caller of the remote method actually call these methods. Any application that wants to make a remote method call must share the interface definition with the application that implements the interface. In the implementation point of view, that means that the calling application and the application that implements the remote interface have to be compiled with the same AIDL files. [4]

When a client (such as an activity) calls `bindService()` to connect to the service, the client's `onServiceConnected()` callback receives the instance of remote interface (IEventPlannerService.Stub), returned by the service's onBind method as described in previous section. When the client receives the IBinder in the `onServiceConnected()` callback, it must call YourServiceInterface.Stub.asInterface(service) to

cast the returned parameter to YourServiceInterface type. The following example code is taken from EventPlanner application.

```
private ServiceConnection serviceConn =
new ServiceConnection() {
    @Override
    public void onServiceConnected(
          ComponentName name,
          IBinder service) {
          mEventPlannerService =
          IEventPlannerService.Stub.asInterface(service);
    }

    @Override
    public void onServiceDisconnected(ComponentName name) {
          mEventPlannerService = null;
    }
};
```

The parameter named service in *`onServiceConnected()`* method is a reference to an IBinder interface. The Binder abstract class implements *`IBinder`*, and the Stub class (AIDL generated class) extends `Binder`.

The Proxy class is the counterpart of the Stub abstract class. Looking inside the Proxy class in IEventPlannerService.java file, it has methods that have the same signature as the remote methods defined in the AIDL file. Here, unlike in the abstract class Stub, the methods are implemented, and the implementations create Parcel objects and fill them with the "flattened" parameters in exactly the right order for the `onTransact()` method of stub class to "unflatten" them and call the remote methods.

That means an application calls a remote method by getting an instance of the Proxy class and calling the remote methods as if they were local. This is shown in below code taken from the SelectContactsActivity class.

```
mEventPlannerService.sendNewEvent(
          phoneNumberList,
          eventData);
```

Recall that `mEventPlannerService` is returned from the `IEventPlannerService.Stub.asInterface` method. Because the caller gets a Proxy object and the remote methods are implemented in a Stub object, and because both Proxy and Stub implement IEventPlannerService, it all looks like a local method call, but the implementations of the methods are completely different in the calling application and the application that implements the remote methods.

The application calling the remote interface gets an instance of the Proxy class. The instance also implements "proxy" methods with the same signature as the remote methods, but they package up their parameters into a Parcel object and send them off to the application that implements the remote methods and unpackages and returns the results. In the remote application, a concrete class extending Stub has implementations of the remote methods. The onTransact method "unflattens" data in a Parcel object, calls the remote methods and "flattens" the result, writes it into a Parcel, and sends that Parcel object back to the calling application.

4.3. Intents

Intents provide highest level abstraction of IPC in Android. `Intent` objects deliver data from one application to another, providing a simple and convenient form of IPC [4]. Three of the four Android application component types: activities, services, and broadcast receivers are activated by intent. Intent messaging is a facility for late runtime binding between components in the same or different applications. The intent itself, an `Intent` object, is a passive data structure holding an abstract representation of an operation to be performed. "Abstract" means that the performer of the desired operation does not have to be defined in the intent. [13, 28]

When activating activities and services, intent defines the action to perform for example, to “open contacts”, “show image” or “send” something and may specify the URI of the data to act on. An activity can be started to receive a result, in which case, the activity also returns the result in `Intent`. For example, intent can be issued to let the user pick a phonebook contact and have it returned. The return intent includes a URI pointing to the chosen contact. The `intent` contains `action` and a `data` field as main information. The following code shows example of intent.

```
Intent intent = new Intent(
          Intent.ACTION_PICK,
          ContactsContract.Contacts.CONTENT_URI);
startActivityForResult(intent, PICK_CONTACT);
```

This intent is delivered by the intent reference monitor to the Binder that is assigned to the action `ACTION_PICK` with URI `ContactsContract.Contacts.CONTENT_URI`. This will open the phone book contacts and returns a contact picked by the user.

Activity can be started by passing an Intent to `startAcitivity()` or `StartActivityForReuslt()`. Service can be started by passing an Intent to `startService()` or you can bind to the service by passing an Intent to `bindService()`. A broadcast can be initiated by passing an Intent to methods like `sendBroadcast()`, `sendOrderedBroadcast()`, or `sendStickyBroadcast()`. The `Intent` class itself provides constructors, accessors, and other utilities for han-

dling the content of an `Intent` object. An important set of accessors are those named `putExtra`. This is used attach "extra" data to an Intent. This data can be used for general purpose inter process communication. [28]

There are two forms of intents. An explicit intent addresses to a specific component. On the other side, an implicit intent gives the decision to the Android system, which component is addressed. If multiple components for one purpose are installed, the system will choose the best component to run the intent.

4.4. Binder

This section describes the origin of Binder, facilities and communication model offered by Binder.

4.4.1. Origin

The Binder was originally developed under the name OpenBinder by Be Inc. and later Palm Inc. under the leadership of Dianne Hackborn. Its documentation claims OpenBinder as “... a system-level component architecture, designed to provide a richer high-level abstraction on top of traditional modern operating system services.” The Binder has the facility to provide bindings to functions and data from one execution environment to another. The OpenBinder implementation runs under Linux and extends the existing IPC mechanisms. [29]

Binder in Android is a modified implementation of OpenBinder. In [30] Dianne Hackborn states that “In the Android platform, the binder is used for nearly everything that happens across processes in the core platform." Android Binder implementation is available in the kernel source at `drivers/misc/binder.c`, with include file `include/linux/binder.h`.

4.4.2. Facilities

Based on [31, 32, 33], methods on remote objects can be called as if they were local object methods. This is achieved with a synchronous method call. A pool of threads is associated to each service application to process incoming IPC. Binder performs mapping of object between two processes and uses an object reference as an address in a process’s memory space.

The developer can create and use the remotable object by sub-classing the Binder class available in android.os package. Binder class is an implementation of IBinder which is base interface for a remotable object, the core part of a lightweight remote procedure call mechanism designed for high performance when performing in-process and cross-process calls.

In most of the situations developers would not need to implement Binder class directly, instead using the AIDL tool to describe the desired interface, having it generate the appropriate Binder subclass. However, developer can derive directly from Binder to

implement custom RPC protocol or simply instantiate a raw Binder object directly to use as a token that can be shared across processes. [31]

AIDL has the feature that an application does not need to know if a service is running in a server process or in the local process. Android's application concept makes it possible to run a service either in an own process or in the activity process. This makes it easy for an application developer to export services to other Android applications without rewriting the code. [29]

4.4.3. Communication model

The Binder framework communication is a client server model. A client will initiate a communication and wait for response from a server. The Binder framework uses a client side proxy for communication. On the server side, a thread pool exists for working on requests. In *Figure 4.3* the process A is the client and holds the proxy object which implements the communication with the Binder kernel driver. Process B is the server process, with multiple Binder threads. The Binder framework will spawn new threads to handle all incoming requests, until a defined maximum count of threads is reached. The proxy objects are talking to the Binder driver that will deliver the message to the destinated object. [29]

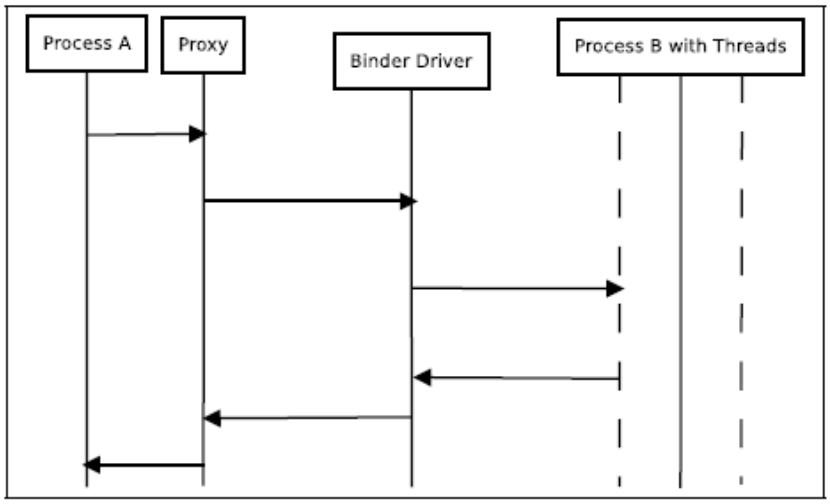

Figure 4.3. Binder Communication [29]

The Binder driver copies the transmission data from the user memory address space of the sending process to its kernel space and then copies the transmission data to the destination process. This is achieved by the copy from user and copy to user command of the Linux kernel. The data transaction is presented in *Figure 4.4*.

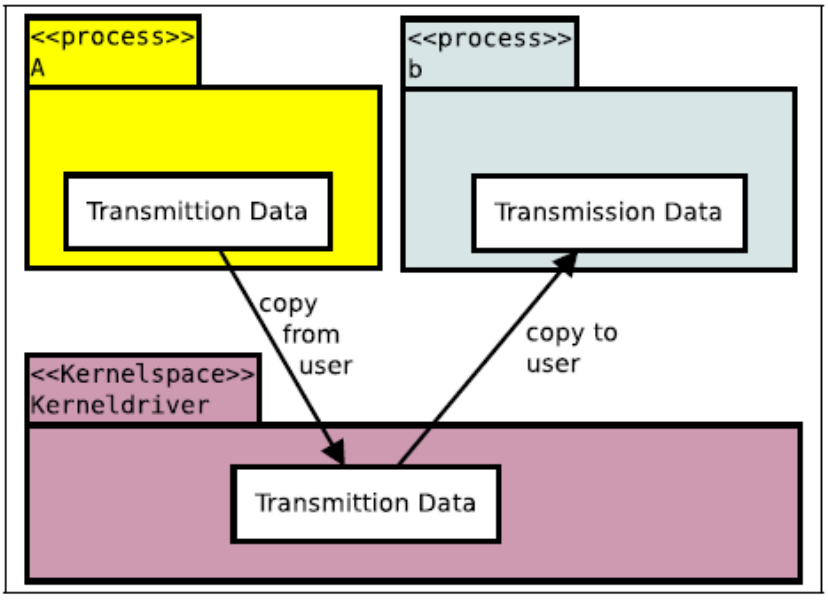

Figure 4.4. Binder Transaction [29]

5. DEVELOPMENT OF APPLICATION

This chapter describes the functionality of Event Planner application, its architecture, design, and implementation.

5.1. Application functionality

With EventPlanner application user can create and send event with voting options about places and different times. Users with EventPlanner can answer to received events and also see final decision and voting results. Planner of the event can make decision about the event and send result to all participants. *Figure 5.1* shows the use cases in the application.

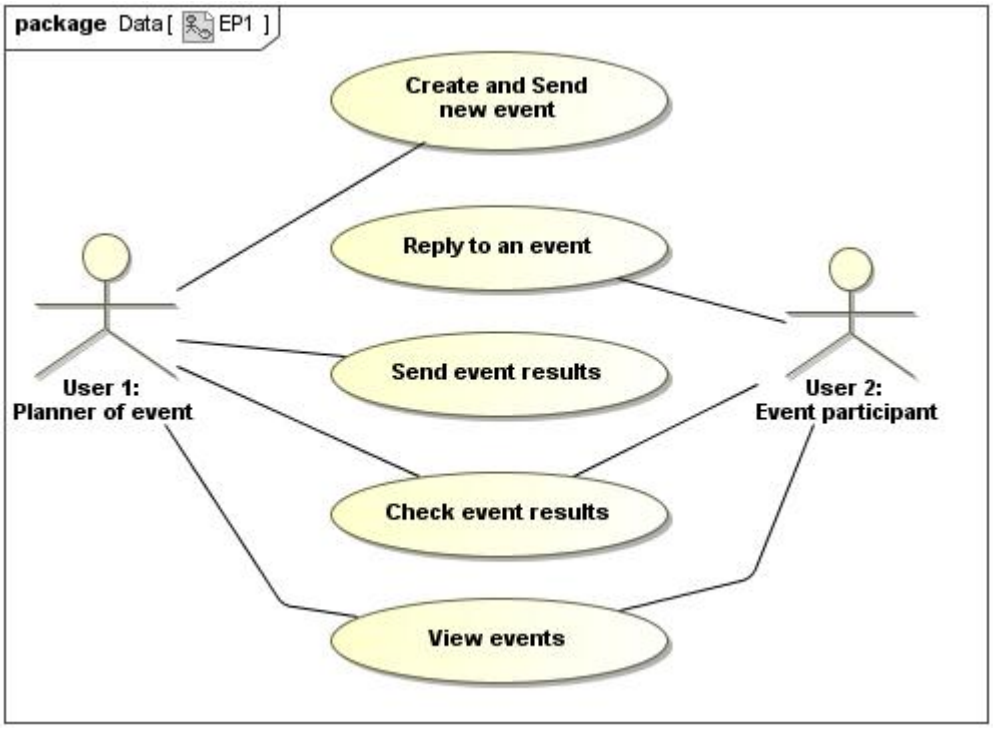

Figure 5.1. EventPlanner Use-case Diagram

Create and send new event: The user (planner of the event) creates new event by giving a name to the event, for e.g. dinner, enters maximum two options for the place and two options for the date and times. Then user selects participants from phonebook contacts and select send event button from UI to send the event details to selected participants via SMS.

View events: Application provides three main views to view the current events. One view shows all the events planned by the current user him/her-self. One view shows events received from friends and which are yet to be answered by the current user. And another view shows list of events which have been confirmed by the planner of the event.

Reply to an event: When a user receives a new event, user can select the attendance "yes" or "no" and selects preferred place and time if answer is "yes". Then selects "send answer" button from UI to send the answer via SMS.

Send event results: After receiving event answers from the participants, the planner of the event can check the answers and votes received from participants and makes the decision by selecting "yes" or "no" to the event and selects one place and one date/time options if decision is "yes". Then user selects "send result" button from UI to send the event confirmation to all participants.

Check event results: The planner of the event and participants can view the result of the events in a detailed view.

5.2. Application architecture

Event Planner application consists of client component which implements user interface (Android Activities) and a service component, running in a separate process in background doing the actual SMS communication and database management for the application. The application architecture diagram shown in *Figure 5.2* has the following features.

- Service runs in a separate process, letting Android platform to manage its lifecycle and resources separately from client.
- Service runs all the time in the background irrespective of whether the client is running currently or not.
- The client (activities) runs in a separate process, sends requests to service, gets results from service, controls the service and updates the UI.
- Since the service runs in separate processes, client uses IPC to communicate with service.
- Service does all the needed logic of sending the event SMSs to participants, receiving event replies, confirmations from other users via SMS communication and maintains local database of events.
- Service can be configured to start during system boot-up.

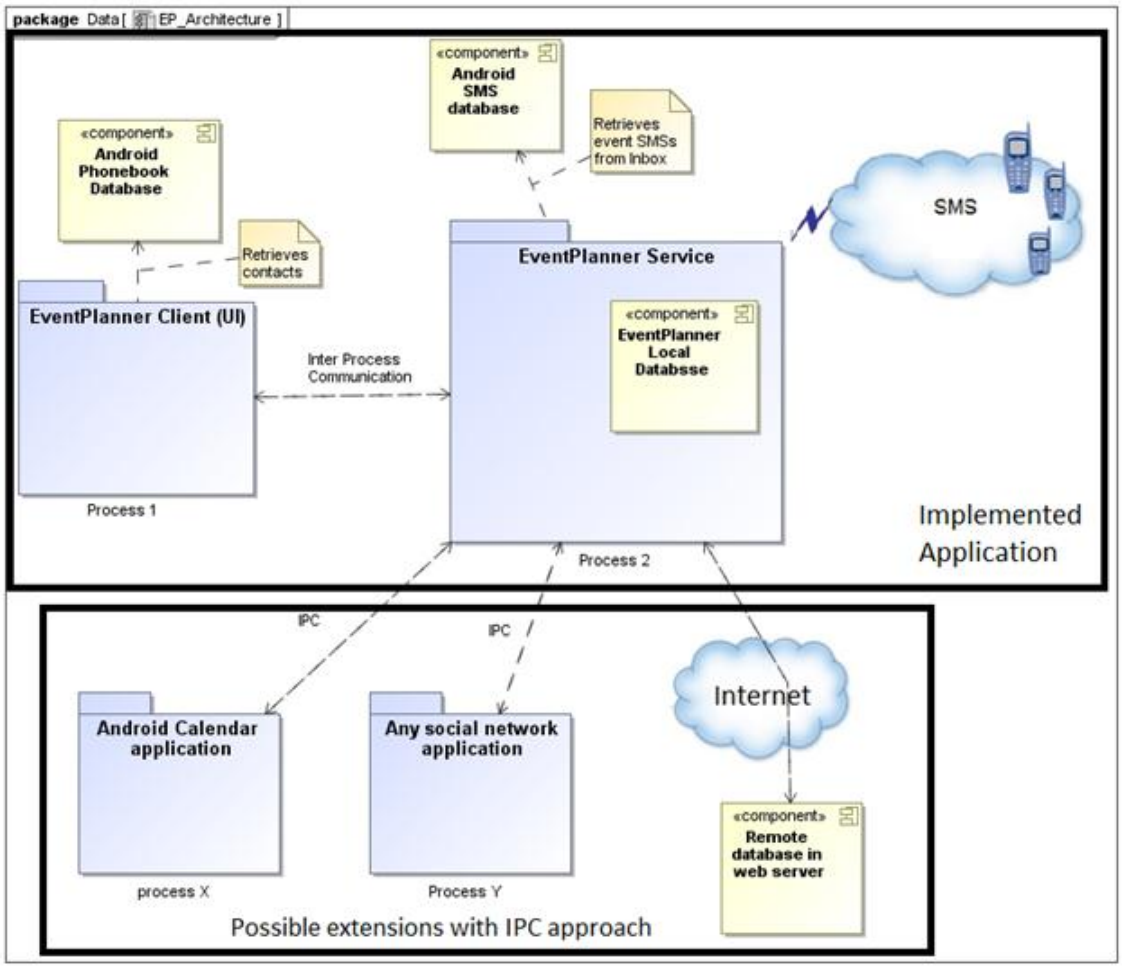

Figure 5.2. Application architecture

Architecture diagram shows two parts. One is the implemented application to study the IPC mechanism. Bottom part of diagram gives an idea about how to extend the application so that some other applications could use Event Planner service using IPC to perform similar tasks.

5.3. Application design

The Event Planner client and service components are designed as a separate java packages. In the following section, these components are explained using the UML diagrams. The client and service parts are modeled by separate UML activity partitions.

5.3.1. Event Planner client

In *Figure 5.3* the event planner client class diagram is presented. The activity class **MainView**, derived from TabActivity is the entry point to the application. MainView starts EventPlanner service explicitly from `onCreate()` method. MainView provides a menu option to create and send a new event. **CreateNewEventActivity** shows UI for taking user input for event name, place options, time options. From this activity user

moves to **SelectContactsActivity** to select the contacts from phone book and then sends the event. SelectContactsActivity bind to EventPlannerService and gets instance of IEventPlannerService interface and sends new event by calling the remote method `sendNewEvent()`.

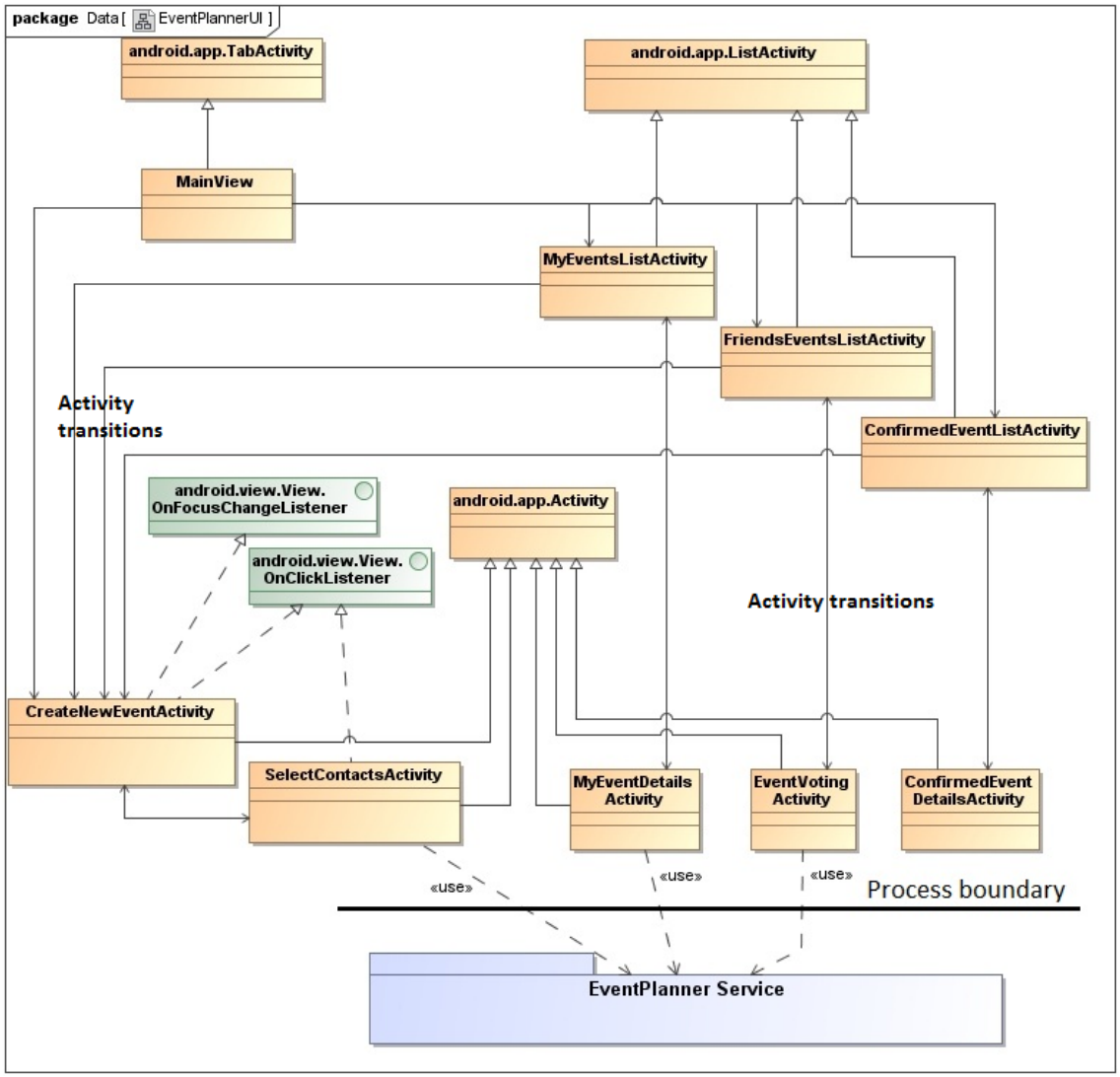

Figure 5.3. EventPlanner client class diagram

MainView features three tabbed views that show the list of events. **MyEventsListActivity** shows list of new events planned by the current user. **FriendsEventsListActivity** shows list of new events received form friends. **ConfirmedEventListActivity** shows list of events confirmed by the event organizer.

From each of the list activities user can navigate to event details activities. **EventVotingActivity** shows list of participants, voting options for place and time. User can choose from the options and send answer by calling the remote method `sendEventReply()` using IEventPlannerService instance. **MyEventDeatilsActivity** shows event details and provides options to choose the event confirmation, place and time of the event and sends the confirmation by calling remote method `send-`

`EventConfirmation()`. Finally **ConfirmedEventDetailsActivity** shows confirmed event details.

5.3.2. Event Planner service

In *Figure 5.4* the EventPlanner service class diagram is presented.

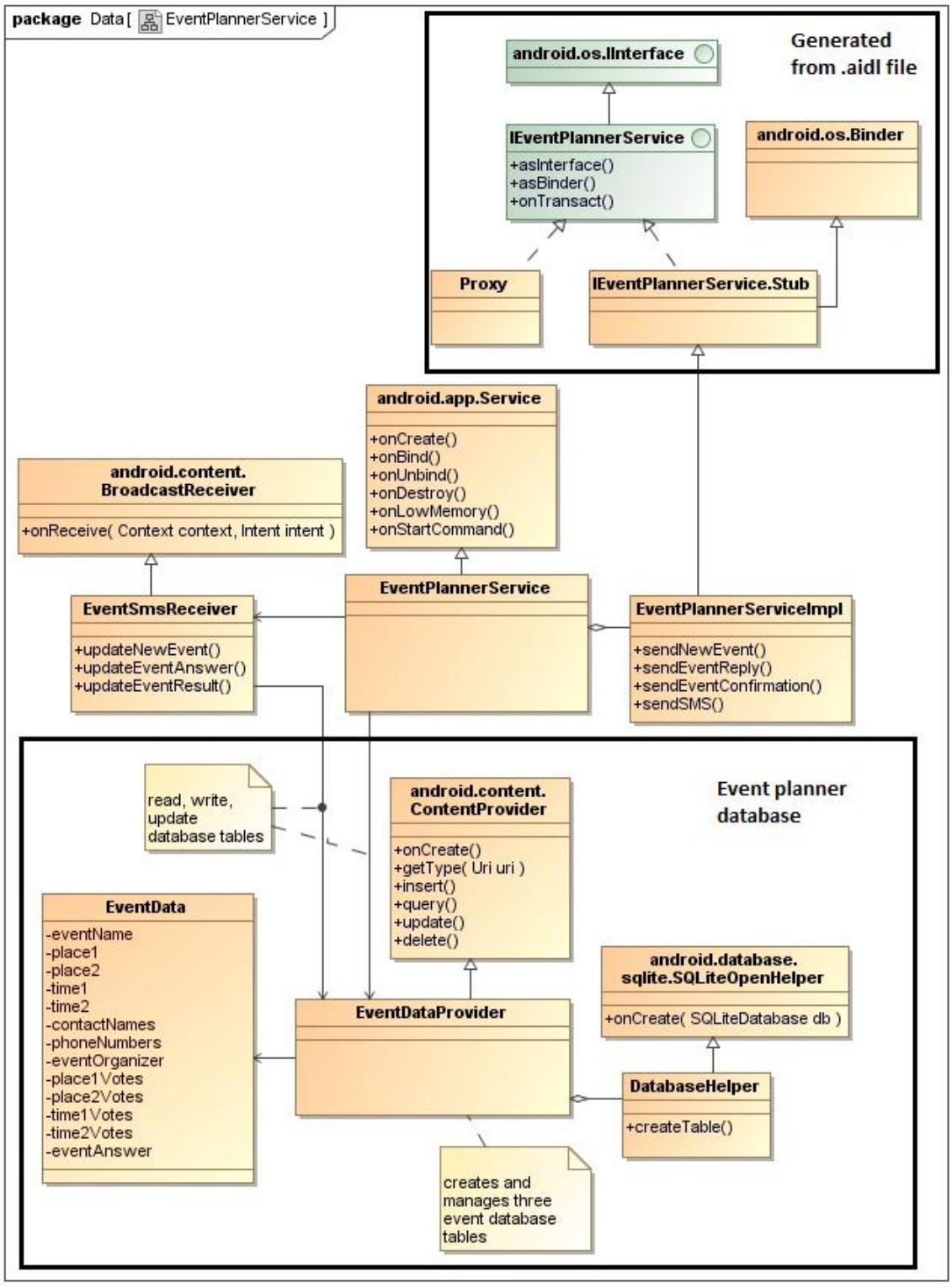

Figure 5.4. EventPlanner service class diagram

The class **EventPlannerService**, derived from `android.app.Service` is the main class in the service side of this application. EventPlannerService class overrides the lifecycle callback methods `onCreate()`, `onDestroy()`, `onBind()`, `onUnbind()`,`onStartCommand()`.

The **EventPlannerServiceImpl** is another important class which actually implements the interface generated the by AIDL. EventPlannerServiceImpl is derived from IEventPlannerService.Stub class and inherits `android.os.Binder`. EventPlannerServiceImpl class implements the remote methods `sendNewEvent()`, `sendEventReply()` and `sendEventConfirmation()`. This class is designed to use SMS service to send the event data to the selected contacts. This class uses two pending intents to keep track of the SMS delivery status. *Figure 5.4* shows the classes generated by the AIDL. The AIDL generated code in IEventPlannerService.java file is given in APPENDIX 2.

Another part of EventPlanner Service is EventSmsReceiver class. **EventSmsReceiver** inherits `android.content.BroadcastReceiver` to receive incoming SMS. In `onReceive()` method looks for event planner SMSes and process the SMS. Based on received event SMS this class updates the database tables.

Finally EventPlanner service manages its own local database tables to store all the events. For this purpose EventPlanner service uses **EventDataProvider** which is a ContentProvider class. EventDataProvider class uses **DatabaseHelper** class to create tables in the database and performs database operations like inserting new events, querying, table updates, deleting the events from tables.

5.4. Application implementation

In the following section, first there will be an explanation about the tools used for developing the application. Secondly there will be description of manifest files of event planner client and service packages followed by User Interface of the developed application.

5.4.1. Development tools

The development of EventPlanner application has been made over Ubuntu 10.04 LTS (Lucid Lynx) with 2.6.32 Linux kernel [34]. Eclipse 3.6.1 (Helios) (IDE version 1.3.1) is used with the addition of the ADT plugin version 10.0.1. Application has been developed with Android SDK 2.2 (API level 8). In addition following tools have been used during the application implementation.

The **Android emulator** [35] is a virtual mobile device that runs on the development machine. The emulator allows developer to develop, debug and test the Android application without using a real device.

The **Android Debug Bridge** (ADB) [36] is a client-server application that lets developers communicate with an emulator instance or connected Android-powered device and install .apk application files. It consists of three components:

- A client, which runs on the development machine. Client can be invoked from a shell by issuing an adb command.
- A server, which runs as a background process on the development machine and works as a link between the client and the adb daemon running on an emulator or device.
- A daemon, which runs as background process on each emulator or device instance.

The **Dalvik Debug Monitor Server** (DDMS) [37] is a debugging tool integrated with DVM. DDMS is integrated into Eclipse and is also shipped in the `tools/` directory of the SDK. It works with both, the emulator and a connected device. DDMS also provides port-forwarding services, screen capture on the device, thread and heap information on the device, logcat, process, and radio state information, incoming call and SMS spoofing, location data spoofing, exploring the file-system.

The **LogCat** [38] tool is used to see the debug logs from various applications and portions of the system. Logs are collected in a series of circular buffers, which then can be viewed and filtered by the logcat command. Logcat can also be used from an ADB shell to view the log messages.

Android Virtual Devices (AVDs) are configurations of the emulator options formed with a hardware profile, a map to the system image, a dedicated storage area on the development machine, and other options.

Android Asset Packaging Tool (aapt) constructs the distributable Android package files (.apk) including the resources.

5.4.2. Manifest files

Event Planner client and service packages use different Android manifest files. The manifest consists of a root `<manifest>` tag with a package attribute set to the project's package. The `<manifest>` tag includes nodes that define the application components, security settings, and requirements that make up the application. All the activities in the client are placed as child items in the `<application>` tag. The debuggable attribute is set to true in the application tag to enable debugging on real device. The icons and the labels are referred to the resources folders with an at-sign (@).

Both client and service use the minimum SDK version 8. This is specified inside the `<uses-sdk>` tag. If minimum version is not specified, system will assume one and the application will crash if it attempts to access APIs that are not available on the host device. `<uses-permission>` is another important manifest tag. This tag declares the permissions that are needed for the application function properly. Event planner client uses `android.permission.READ_CONTACTS` permission to be able to open and select contacts from contacts database. Event planner service uses `SEND_SMS, RECEIVE_SMS, WRITE_SMS, READ_SMS` permissions to be able to perform SMS operations. The manifest of EventPlanner Service specifies EventPlannerService class in `<Service>` tag. Service tags also support `<intent-filter>` child tags to allow

late run-time binding. EventSmsReceiver class is specified in <receiver> tag and EventDataProvider class is specified in <provider> tag.

5.4.3. User interface

EventPlanner application has been mainly tested on Emulator as well as real devices, a LG-P500 (Android version 2.2.1) and Samsung Galaxy phones. UI has been implemented for portrait layout. All the activities have been implemented with common UI elements like text labels, editable fields, buttons, and images. The following Figures show the EventPlanner application screenshots from emulator.

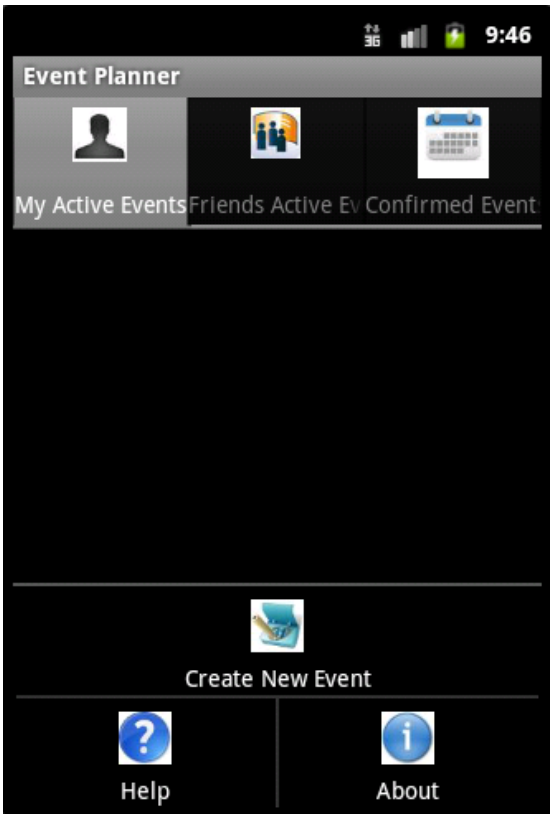

Figure 5.5. Event list is empty

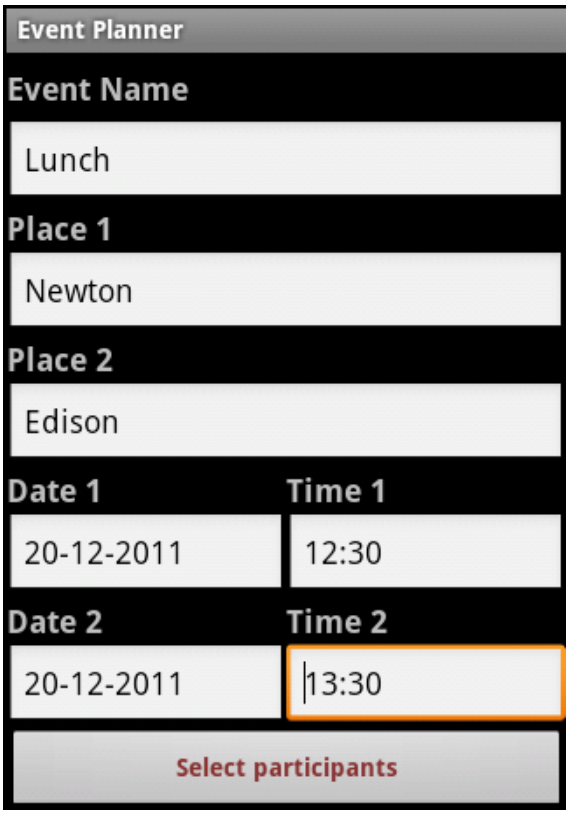

Figure 5.6. Creating a new event

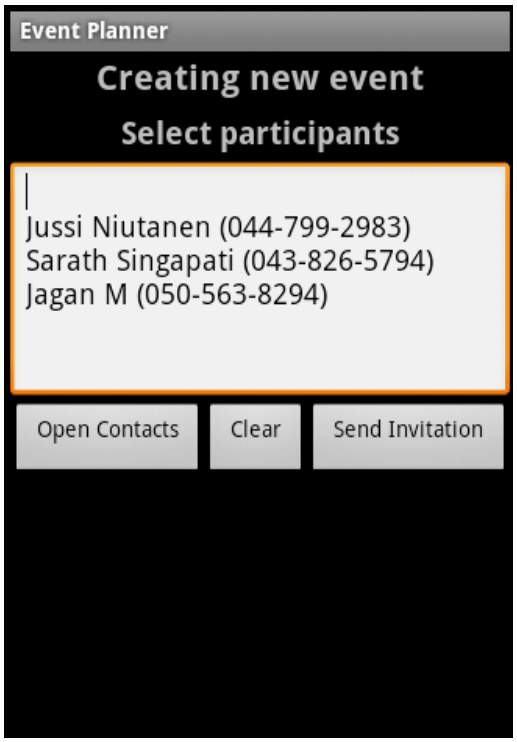

Figure 5.7. Selecting contacts

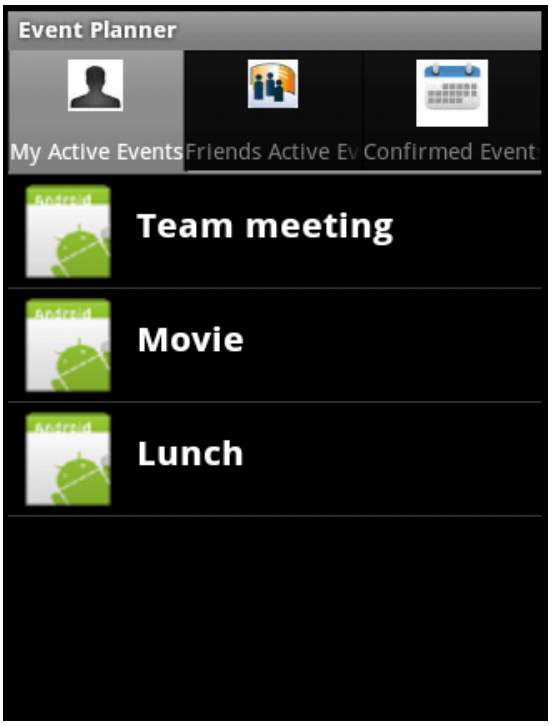

Figure 5.8. List of events

Figure 5.9. Event Details View

Figure 5.10. Confirmed Event details

6. EVALUATION

In this thesis, we are interested in how to use inter process communication in Android platform, different available mechanisms, platform and tools support for developing applications that communicate using IPC.

6.1. Android platform and tools

The Android platform is a combination of Linux kernel and a Java application interface. The Linux kernel is very well tested and used over many years. In Android, Linux kernel has been slightly modified to suit the mobile environment. Linux operating system is highly stable and having the same makes the Android platform robust. Android platform is lightweight and is designed for low memory overhead, limited resources, reduced power consumption and secure applications. For the developers the Java platform is most suitable for developing applications, and object oriented language increases the usability and maintainability of the system.

Android has a well-engineered development environment. The Android SDK supports several different integrated development environments (IDEs). Eclipse is the IDE that is best integrated with the Android SDK. It has all the needed tools and the ADT plugin for Eclipse provides easy application development experience. Applications can be tested and debugged either on a real Android device or on an emulator. For most developers, using an emulator is easier for initial development and debugging, followed by final testing on real devices.

6.2. IPC support in Android platform

Based on the implementation of the sample application, we can say that Android platform is suited to host a variety of applications and to maximize user choice. The platform is intended to eliminate the duplication of functionality in different applications by reusing functionality among applications. Thus, IPC is the most effective, scalable and correct way to connect Android components to each other.

Android's Intent mechanism is independent of specific application implementations. In an Android application, you do not say you want to open a web page through a specific application; instead, you say you want to open a web page through whatever application is available. The operating system takes care of figuring out what application can open web pages, starts that application if needed, and connects your request so the web page can be opened. The user can substitute different MP3 players, or different email clients at will, and Android adapts automatically.

Android's remote method system is a powerful feature with many uses, but remote method calls are overkill in many cases. When designing applications, first consider whether inter process communications needs fit what Intents and the `Context` class's Intent related methods can do. When using IPC to provide a user interface in an Activity, Intent mechanism is easy to use and appropriate to the task. For example, "Event-Planner" application uses Intent mechanism to open and select phonebook contacts from contacts application.

6.3. Choosing interface definition

When creating a service that provides binding and performs IPC, as explained already, an `IBinder` must be provided to the clients that provide the programming interface to interact with the service. Developer can choose how to define the interface based on the application requirements and application components.

If your service is used only by the local application and runs in the same process as the client, then developer can create the interface by extending the `Binder` class and returning an instance of it from `onBind()`. The client receives the `Binder` and can use it to directly access public methods available in `Binder` implementation and in the service. The reason the service and client must be in the same application is so the client can cast the returned object and properly call its APIs. The service and client must also be in the same process, because this technique does not perform any marshaling across processes. This is the preferred technique when your service is merely a background worker for your own application. For example, this would work well for a music application that needs to bind an activity to its own service that is playing music in the background**.**

When the service is used by other applications and you want your service to handle multiple requests simultaneously then AIDL should be used. AIDL performs all the work to decompose objects into primitives that the operating system can understand and marshal them across processes to perform IPC. In this case, your service must be capable of multi-threading and be built thread-safe and it can result in a more complicated implementation.

Another choice is using a `Messenger` object available in `android.os` package. `Messenger` provides message-based communication across processes. The service defines a `Handler` that responds to different types of `Message` objects. This `Handler` is the basis for a `Messenger` that can then share an `IBinder` with the client, allowing the client to send commands to the service using `Message` objects. Additionally, the client can define a `Messenger` of its own so the service can send messages back. The `Messenger` creates a queue of all the client requests in a single thread, so the service receives requests one at a time. This has to be investigated and confirmed in future work. [39, 40]

6.4. Correct way to bind to service

Based on the implemented Service and Activity classes in “EventPlanner” application, some of the important findings about binding to a service are listed below.

You should always trap `DeadObjectException`, which are thrown when the connection has broken. This is the only exception thrown by remote methods. This means that the object you are calling has died, because its hosting process no longer exists. This happens when the service has been stopped already either killed by the OS, or stopped from your application.

Binding and unbinding to service should usually be performed during corresponding start-up and tear-down moments of the client's lifecycle. For example, If it is needed to interact with the service while activity is visible, you should bind during `onStart()` and unbind during `onStop()`. If the activity need to receive responses even while it is stopped in the background, then you can bind during `onCreate()` and unbind during `onDestroy()`. This implies that your activity needs to use the service the entire time it is running even in the background.

Binding and unbinding should not be done during activity's `onResume()` and `onPause()`, because these callbacks occur at every lifecycle transition. This way the processing that occurs at these transitions can be kept in minimum. Also, if multiple activities in your application bind to the same service and there is a transition between two of those activities, the service may be destroyed and recreated as the current activity unbinds (during pause) before the next one binds (during resume).

6.5. Binder evaluation

Binder is simple, performant and can be used in performance-critical parts. Binders are not ideal for transferring large data streams (like audio/video) since every object has to be converted to and back from a Parcel. Binder does not manage version information. This means APIs built on Binder should remain compatible with older versions if the APIs are open for other applications to use, and it means that consumers of remote APIs should be resilient to failures caused by incompatibilities. These exceptions should be handled.

7. CONCLUSIONS

The aim of this thesis work was to study the Android platform to find out its possibilities for developing applications that provide services and communicate with each other using Inter Process communication mechanisms. As part of the study, an application was developed using many of the Android's building blocks and AIDL in particular to implement IPC. This work resulted in a fully functional application with a good structure for a background Service with Activities connected via IPC. This approach can be used for developing various applications.

The ability to develop a Service that runs in the background and does something when the user is not actively working with the application is a great option for application developers. Examples of usage include email or IM applications that send you a notification whenever there is a new message, background music players, background downloads etc. IPC is the effective, scalable and right way to connect Android components to each other. But application developers should keep in mind that too many services doing too much work in the background will slow down the phone and consume the battery.

Android applications could avoid inter process communication and provide functions in packages loaded by the applications when needed. If applications had to exchange data, they could use the file-system or other traditional Unix/Linux IPC mechanisms like sockets, shared memory, etc. But these practices are error prone and hard to maintain. In almost all cases, a remote procedure call is efficient enough to make it a better alternative to loading a library, especially one that dynamically allocates a significant amount of memory into the virtual machine's address space. And if a process exposing an RPC interface fails, it is less likely to bring down the Android UI with it.

As a result, there has been a need for more robust component-like systems. Intents and remote methods suits well for Android. In summary, this thesis suggests that inter process communication is very useful mechanism in Android programming model to provide flexible, efficient message passing between processes and Android platform has all the needed tools, operating system components to implement IPC.

REFERENCES

[1] Gartner. Press releases, September 10, 2010.
http://www.gartner.com/it/page.jsp?id=1434613
Visited on 10.05.2012

[2] Openhandset Alliance.
http://www.openhandsetalliance.com/index.html
Visited on 03.01.2012

[3] Google Blog.
http://googleblog.blogspot.com/2007/11/wheres-my-gphone.html
Visited on 03.01.2012

[4] Android Application Development, 1st edition, by Rick Rogers, John Lombardo.
Publisher: O'Reilly Media, Inc.

[5] Professional Android 2 Application Development, by Reto Meier
Publisher: Wiley Publishing, Inc.

[6] Anatomy & Physiology of an Android, 2008 Google I/O Session.
http://sites.google.com/site/io/anatomy--physiology-of-an-android
Visited on 06.01.2012

[7] Android Kernel Features.
http://elinux.org/Android_Kernel_Features
Visited on 06.01.2012

[8] Android Linux Kernel Additions.
http://www.lindusembedded.com/blog/2010/12/07/android-linux-kernel-additions/
Visited on 07.01.2012

[9] Android Logger.
http://www.elinux.org/Android_logger
Visited on 03.02.2012

[10] Android Power Management.
http://www.elinux.org/Android_Power_Management
Visited on 03.02.2012

[11] What is Android?
http://developer.android.com/guide/basics/what-is-android.html

Visited on 15.01.2012

[12] Dalvik Virtual Machine.
http://www.dalvikvm.com/
Visited on 17.01.2012

[13] Android Application Fundamentals.
http://developer.android.com/guide/topics/fundamentals.html
Visited on 15.01.2012

[14] Inside the Android Application Framework, 2008 Google I/O Session.
http://sites.google.com/site/io/inside-the-android-application-framework
Visited on 25.01.2012

[15] Android Activities.
http://developer.android.com/guide/topics/fundamentals/activities.html
Visited on 26.01.2012

[16] Android Service.
http://developer.android.com/guide/topics/fundamentals/services.html
Visited on 15.03.2012

[17] Android Content Provider.
http://developer.android.com/guide/topics/providers/content-providers.html
Visited on 03.02.2012

[18] Deep Inside Android…, OpenExpo 2008 – Zurich, Gilles Printemps.

[19] Android Broadcast Receiver.
http://developer.android.com/reference/android/content/BroadcastReceiver.html
Visited on 03.02.2012

[20] Android Manifest File.
http://developer.android.com/guide/topics/manifest/manifest-intro.html
Visited on 19.05.2012

[21] Android Processes and Threads.
http://developer.android.com/guide/topics/fundamentals/processes-and-threads.html
Visited on 15.02.2012

[22] Android Reference Thread.
http://developer.android.com/reference/java/lang/Thread.html

Visited on 15.02.2012

[23] Android Responsiveness.
http://developer.android.com/guide/practices/design/responsiveness.html
Visited on 17.02.2012

[24] Android AsyncTask.
http://developer.android.com/reference/android/os/AsyncTask.html
Visited on 20.02.2012

[25] Android Handler.
http://developer.android.com/reference/android/os/Handler.html
Visited on 20.02.2012

[26] Inter process method invocation in Android, Tetsuyuki Kobayashi, 2011.4.9

[27] Android Interface Definition Language
http://developer.android.com/guide/developing/tools/aidl.html
Visited on 03.03.2012

[28] Android Intents.
http://developer.android.com/guide/topics/intents/intents-filters.html
Visited on 14.04.2012

[29] Android Binder - Android Interprocess Communication, Thorsten Schreiber, October 5, 2011

[30] Dianne Hackborn About Binder.
https://lkml.org/lkml/2009/6/25/3
Visited on 20.04.2012

[31] Android Binder.
http://developer.android.com/reference/android/os/Binder.html
Visited on 21.04.2012

[32] Android Binder.
http://elinux.org/Android_Binder
Visited on 21.04.2012

[33] Binder IPC Mechanism.
http://www.angryredplanet.com/~hackbod/openbinder/docs/html/BinderIPCMechanism.html

Visited on 22.04.2012

[34] LucidLynx Technical Overview.
https://wiki.ubuntu.com/LucidLynx/TechnicalOverview
Visited on 07.04.2012

[35] Using The Android Emulator.
http://developer.android.com/guide/developing/devices/emulator.html
Visited on 07.04.2012

[36] Android Debug Bridge.
http://developer.android.com/guide/developing/tools/adb.html
Visited on 07.04.2012

[37] Using DDMS.
http://developer.android.com/guide/developing/debugging/ddms.html
Visited on 07.04.2012

[38] Android Logcat tool.
http://developer.android.com/guide/developing/tools/logcat.html
Visited on 07.04.2012

[39] Android Messenger Class Reference.
http://developer.android.com/reference/android/os/Messenger.html
Visited on 28.04.2012

[40] Android Bound Services.
http://developer.android.com/guide/topics/fundamentals/bound-services.html
Visited on 28.04.2012

[41] Intent Class Reference.
http://developer.android.com/reference/android/content/Intent.html
Visited on 29.04.2012

APPENDIX 1: EVENT PLANNER MANIFEST FILE

EventPlanner Client Manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.ipc.thesis.android"
    android:versionCode="1" android:versionName="1.0" >
    <uses-sdk android:minSdkVersion="8" />
    <uses-permission android:name="android.permission.READ_CONTACTS" />
    <application android:debuggable="true" android:icon="@drawable/my_events"
        android:label="@string/app_name" >
        <activity android:name="MainView" android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.DEFAULT" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".CreateNewEventActivity" </activity>
        <activity android:name="SelectContactsActivity" > </activity>
        <activity android:name="MyEventsListActivity" > </activity>
        <activity android:name="MyEventDetailsActivity" > </activity>
        <activity android:name="FriendsEventsListActivity" > </activity>
        <activity android:name="EventVotingActivity" > </activity>
        <activity android:name="ConfirmedEventListActivity" > </activity>
        <activity android:name="ConfirmedEventDetailsActivity" > </activity>
    </application>
</manifest>
```

EventPlanner Service Manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.ipc.eventplannerservice"
    android:versionCode="1" android:versionName="1.0" >
    <uses-sdk android:minSdkVersion="8" />
    <uses-permission android:name="android.permission.SEND_SMS" />
    <uses-permission android:name="android.permission.RECEIVE_SMS" />
    <uses-permission android:name="android.permission.READ_SMS" />

    <application android:icon="@drawable/icon" an-
    droid:label="@string/app_name" >
        <service android:name="EventPlannerService" >
            <intent-filter> <action an-
            droid:name="com.ipc.eventplannerservice.IEventPlannerService" />
            </intent-filter>
        </service>
        <receiver android:name="EventSmsReceiver" >
            <intent-filter> <action an-
            droid:name="android.provider.Telephony.SMS_RECEIVED" />
            </intent-filter>
        </receiver>
        <provider android:name="EventDataProvider"
            android:authorities="com.ipc.eventdata.provider.EventData" />
    </application>
</manifest>
```

APPENDIX 2: AIDL GENERATED FILE

```
package com.ipc.eventplannerservice;
public interface IEventPlannerService extends android.os.IInterface
{
/** Local-side IPC implementation stub class. */
public static abstract class Stub extends android.os.Binder implements
com.ipc.eventplannerservice.IEventPlannerService
{
private static final java.lang.String DESCRIPTOR =
"com.ipc.eventplannerservice.IEventPlannerService";
public Stub()
{
this.attachInterface(this, DESCRIPTOR);
}
/** Cast an IBinder object into an-
com.ipc.eventplannerservice.IEventPlannerService interface, generating a
proxy if needed. */
public static com.ipc.eventplannerservice.IEventPlannerService asInter-
face(android.os.IBinder obj)
{
if ((obj==null)) {
return null;
}
android.os.IInterface iin = (an-
droid.os.IInterface)obj.queryLocalInterface(DESCRIPTOR);
if (((iin!=null)&&(iin instanceof
com.ipc.eventplannerservice.IEventPlannerService))) {
return ((com.ipc.eventplannerservice.IEventPlannerService)iin);
}
return new com.ipc.eventplannerservice.IEventPlannerService.Stub.Proxy(obj);
}
public android.os.IBinder asBinder()
{
return this;
}
@Override public boolean onTransact(int code, android.os.Parcel data, an-
droid.os.Parcel reply, int flags) throws android.os.RemoteException
{
switch (code)
{
case INTERFACE_TRANSACTION:
{
reply.writeString(DESCRIPTOR);
return true;
}
case TRANSACTION_sendNewEvent:
{
data.enforceInterface(DESCRIPTOR);
java.lang.String _arg0;
_arg0 = data.readString();
java.lang.String _arg1;
_arg1 = data.readString();
int _result = this.sendNewEvent(_arg0, _arg1);
reply.writeNoException();
reply.writeInt(_result);
return true;
}
case TRANSACTION_sendEventReply:
```

```
{
data.enforceInterface(DESCRIPTOR);
java.lang.String _arg0;
_arg0 = data.readString();
java.lang.String _arg1;
_arg1 = data.readString();
int _result = this.sendEventReply(_arg0, _arg1);
reply.writeNoException();
reply.writeInt(_result);
return true;
}
case TRANSACTION_sendEventConfirmation:
{
data.enforceInterface(DESCRIPTOR);
java.lang.String _arg0;
_arg0 = data.readString();
java.lang.String _arg1;
_arg1 = data.readString();
int _result = this.sendEventConfirmation(_arg0, _arg1);
reply.writeNoException();
reply.writeInt(_result);
return true;
}
}
return super.onTransact(code, data, reply, flags);
}
private static class Proxy implements
com.ipc.eventplannerservice.IEventPlannerService
{
private android.os.IBinder mRemote;
Proxy(android.os.IBinder remote)
{
mRemote = remote;
}
public android.os.IBinder asBinder()
{
return mRemote;
}
public java.lang.String getInterfaceDescriptor()
{
return DESCRIPTOR;
}
public int sendNewEvent(java.lang.String phoneNumberList, java.lang.String
newEventSMS) throws android.os.RemoteException
{
android.os.Parcel _data = android.os.Parcel.obtain();
android.os.Parcel _reply = android.os.Parcel.obtain();
int _result;
try {
_data.writeInterfaceToken(DESCRIPTOR);
_data.writeString(phoneNumberList);
_data.writeString(newEventSMS);
mRemote.transact(Stub.TRANSACTION_sendNewEvent, _data, _reply, 0);
_reply.readException();
_result = _reply.readInt();
}
finally {
_reply.recycle();
_data.recycle();
```

```
}
return _result;
}
public int sendEventReply(java.lang.String phoneNumber, java.lang.String
eventReplySMS) throws android.os.RemoteException
{
android.os.Parcel _data = android.os.Parcel.obtain();
android.os.Parcel _reply = android.os.Parcel.obtain();
int _result;
try {
_data.writeInterfaceToken(DESCRIPTOR);
_data.writeString(phoneNumber);
_data.writeString(eventReplySMS);
mRemote.transact(Stub.TRANSACTION_sendEventReply, _data, _reply, 0);
_reply.readException();
_result = _reply.readInt();
}
finally {
_reply.recycle();
_data.recycle();
}
return _result;
}
public int sendEventConfirmation(java.lang.String phoneNumberList, ja-
va.lang.String eventConfirmationSMS) throws android.os.RemoteException
{
android.os.Parcel _data = android.os.Parcel.obtain();
android.os.Parcel _reply = android.os.Parcel.obtain();
int _result;
try {
_data.writeInterfaceToken(DESCRIPTOR);
_data.writeString(phoneNumberList);
_data.writeString(eventConfirmationSMS);
mRemote.transact(Stub.TRANSACTION_sendEventConfirmation, _data, _reply, 0);
_reply.readException();
_result = _reply.readInt();
}
finally {
_reply.recycle();
_data.recycle();
}
return _result;
}
}
static final int TRANSACTION_sendNewEvent = (an-
droid.os.IBinder.FIRST_CALL_TRANSACTION + 0);
static final int TRANSACTION_sendEventReply = (an-
droid.os.IBinder.FIRST_CALL_TRANSACTION + 1);
static final int TRANSACTION_sendEventConfirmation = (an-
droid.os.IBinder.FIRST_CALL_TRANSACTION + 2);
}
public int sendNewEvent(java.lang.String phoneNumberList, java.lang.String
newEventSMS) throws android.os.RemoteException;
public int sendEventReply(java.lang.String phoneNumber, java.lang.String
eventReplySMS) throws android.os.RemoteException;
public int sendEventConfirmation(java.lang.String phoneNumberList, ja-
va.lang.String eventConfirmationSMS) throws android.os.RemoteException;
}
```